Development and Applications of Multi-layered Genetic Algorithms to Multi-dimensional Optimisation Problems

by

Galina Vladislavovna Kelareva

Submitted in fulfilment of the requirements

for the Degree

of Doctor of Philosophy

University of Tasmania

Australia

February 2003

To the memory of my parents

and to my fantastic family – Lena, Nadia and Michael

Statement of Originality

I hereby declare that this submission is my own work, that no part of this thesis has been accepted or presented for an award of any degree or diploma by the University or any institution, and that to the best of my knowledge, this thesis contains no material previously published or written by another person except where due acknowledgment has been made in the text or in the bibliography.

Galina V. Kelareva

25.02.03

Statement of Authority of Access

This thesis may be made available for loan and limited copying in accordance with the Copyright Act 1968.

Galina V. Kelareva

25.02.03

Abstract

Genetic algorithms represent a global optimisation method, imitating the principles of natural evolution: selection and survival of the fittest. Genetic algorithms operate on a randomly initialised population of potential solutions to a problem. The solutions develop by passing valuable genetic information to succeeding generations.

Genetic algorithms are known as a robust technique suitable for a variety of optimisation problems. However, when applied to complex combinatorial problems with multiple parameters, conventional genetic algorithms are usually slow and ineffective due to the large search space.

This thesis proposes a novel approach to the development of a genetic algorithm and applies this approach to a maintenance scheduling problem in a power generation system. Problem specific knowledge is utilised to divide the problem into several layers, with each layer representing a part of the initial problem. Solutions are progressively developed, with each layer algorithm finding partial solutions that satisfy specified criteria. These partial solutions are then used as building blocks in the next layer, to progressively build up complete solutions.

The resulting multi-layered genetic algorithm is able to concentrate its search efforts in areas where good quality solutions are likely to be present, therefore producing better results than traditional genetic algorithms. Further developments of the multi-layered genetic algorithm are also suggested in this thesis. The algorithm is combined with a local search method, and heuristic rules are used for initialisation of the population. The combined method results in an effective and fast exploration of the problem's search space and is suitable for a variety of optimisation problems.

The proposed algorithm is implemented using MATLAB programming language and tested on a real power generation system. A number of implementation issues, such as

specific chromosome structure and a varying generation gap; interchangeable solutions and gene convergence; weeding out duplicates from the population and reducing the search space without losing the quality of representing the problem domain, are all discussed. Specifics of a local search method and its representation are also examined. Special attention is paid to developing efficient evaluation and neighbourhood exploration procedures.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr. Michael Negnevitsky, for his valuable suggestions and continuing support and advice throughout my research. I would also like to thank the University of Tasmania for the scholarship, and the School of Engineering for creating the educational environment that enabled me to undertake my postgraduate studies. My appreciation is also extended to Dr. Steve Carter for his suggestions on how to better visualise abstract concepts and uninteresting algorithms. Finally, a grateful acknowledgment to Mr. Laszlo Varga, a Senior Expert with Hungarian Power Companies Ltd., for providing data that was used in the research.

On a personal note, I would like to thank my husband, Michael, the proofreader of my first draft, for his painstaking reading, and conscientious and never-ending struggle with my attempts to butcher the English language. I also wish to thank my daughters: Lena, the proofreader of my final draft; and Nadia, the bibliography minder, for their help and patience, not to mention all those dinners.

Contents

List of Figures

List of Tables

List of Notations and Abbreviations

T - time intervals, $i = 1,\ldots,T$

N – number of units, $j = 1,\ldots,N$

C_j - capacity of unit j

M_j - number of weeks required for maintenance of unit j

$S = \sum_{j=1}^{N} C_j$ - installed capacity of the system

P_i - predicted load in week i

$G_i = S - P_i$ - gross reserve in week i

J_i - units scheduled in week i

$R_i = G_i - \sum_{j \in J_i} C_j$ - nett reserve in week i

R^0 - retained reserve parameter

$a = (a_1, a_2, \ldots, a_N)$ - chromosome with N genes

$\mathcal{A}_j$ - gene pool for gene j, $a_j \in \mathcal{A}_j$

$f_l^{obj} = \min\{R_{il} \mid i = 1,\ldots,T\}$ - objective function of individual l

F_l - fitness of individual l

R^0 - retained reserve parameter

K – number of layers, $k = 1,\ldots,K$

$J^{(k)}$ - units in layer k

$N^{(k)}$ - number of units in layer k

$s = (a_1, a_2, \ldots, a_m)$ - sub-schedule with m units

$L_i = \sum_{j=1}^{m} C_j X_{ij}$ - additional load on the system

$$X_{ij} = \begin{cases} 1, \text{ if } j \in J_i \\ 0, \text{ otherwise} \end{cases}$$

$\mathcal{S}^{(k)}$ - gene pool containing sub-schedules found in layer k

$s_l^{(k)} = \left(a_1, \ldots, a_{N^{(k)}}, s^{(k-1)}\right)$, $a_j \in \mathcal{A}_j$, $s^{(k-1)} \in \mathcal{S}^{(k-1)}$ - sub-schedule l from layer $k>1$

$\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3$ - termination criteria

t – generation

$P(t)$ - population at generation t

$p \in P(t)$ - individual from the population $P(t)$

$F(t)$ - fitness of the population at generation t

Ng - number of genes in an individual

Nc - number of changing genes in an individual

Ns - number of neighbours sampled during local search

$\mathcal{N}_r^{Nc}(x)$ - neighbourhood of x with radius r and Nc changing genes

GA – genetic algorithm

PMX – partially matched crossover

TSP – travelling salesman problem

JSSP – job shop scheduling problem

MLGA – multi-layered genetic algorithm

LS – local search

GLS – genetic local search

MLGLS – multi-layered genetic local search

List of Publications

1 Kelareva, G., Negnevitsky, M., "Multi-layered genetic algorithms for maintenance scheduling in power systems", Proc. Of the 2nd IASTED Int. Conf. On Power and Energy Systems, (Crete, Greece, June 2002), pp 32-37.

2 Kelareva, G., Negnevitsky, M., "Multi-layered genetic algorithm for maintenance scheduling with multiple parameters", Australian Journal of Intelligent Information Processing Systems, Vol.7, No.3/4, 2001, pp. 122-131.

3 Kelareva G., Negnevitsky M., "Multi-layered genetic algorithm for maintenance schedule optimisation", Proc. Australasian Universities Power Engineering Conf. (AUPEC, Perth, WA, Australia, 2001), pp 379-384.

4 Negnevitsky, M., Kelareva, G., "Genetic algorithms application to scheduling problems in power systems", Proc. Of the 1st Japanese-Australian Joint Seminar on Applications of Electromagnetic Phenomena in Electrical and Mechanical Systems (Adelaide, Australia, March 2000), pp. 123-129.

5 Negnevitsky, M., Kelareva, G., "Maintenance scheduling in power systems using genetic algorithms", Proc. Of International Conference on Electric Power Engineering (Budapest, Hungary, 1999), BPT99-445-16.

6 Negnevitsky, M., Kelareva, G., "Genetic algorithms for maintenance scheduling in power systems ", Proc. Of Australasian Universities Power Engineering Conference (AUPEC, Darwin, Australia, Sept. 1999), pp 184189.

7 Negnevitsky, M., Kelareva, G., "Application of genetic algorithms for maintenance scheduling in power systems ", Proc. Of 6th Int. Conf.on Neural Information Processing (ICONIP, Perth, WA, Nov. 1999), pp 447-451.

8 Kelareva, G., Negnevitsky, M., Carter, S., "Application of a fuzzy neural network for predicting air quality", Proc. Int. Conf. On Artificial Intelligence in Science and Technology (AISAT, Hobart, Australia, Dec. 2000), pp 169-173.

Preface

Genetic algorithms (GAs) belong to the group of optimisation methods simulating natural evolution and are applied in various areas, such as function optimisation, strategy planning, scheduling. Their undoubtable strength is the ability perform a global search in the problem domain in parallel. However, GAs are often considered as slow and inefficient when dealing with a real life optimisation problem with multiple parameters and a vast search space. Current research in GAs is concerned with finding the ways to increase the search efficiency when dealing with such problems.

This thesis proposes a new GA technique to provide an efficient optimisation method suitable for a problem with a large search space and vaguely defined constraints. As a case study, a scheduling optimisation problem in a power generating system is considered.

The thesis consists of six chapters. Chapter 1 gives an overview of GAs as an optimisation method and its parameters, with special attention paid to various representation techniques and corresponding genetic operators.

Chapter 2 describes the case study and examines the traditional GA approach to the problem with two different ways of representation, direct and indirect. As a result of discussion about advantages and shortcomings of both representations, the direct representation is selected for further study due to its ability to provide the complete coverage of the problem domain. However, as shown in Chapter 2, a traditional GA that uses a direct representation of the scheduling problem often cannot find any solutions of a good quality due to the overly large search space.

In Chapter 3 the candidate proposes a new optimisation method, a multi-layered genetic algorithm (MLGA), which provides a better chance to explore the problem domain. The new method employs the idea of a partial separability of the scheduling problem by dividing the problem into layers and solving the resulting subsequent problems one after another, gradually building a solution to the initial large problem. The partial schedules

found in a layer are later used as building blocks in the subsequent layer when building new larger schedules. The MLGA gives an opportunity to explore the entire problem domain more efficiently than a traditional GA, however, the new algorithm is still rather slow.

Chapters 4 and 5 further develop the algorithm. In Chapter 4, an MLGA was combined with a local search method, which dramatically improved and fastened the search. In order to do this, the candidate considers several neighbourhood definitions and suggests the most suitable for the problem in question. Local search parameters are identified and fine-tuned in a series of experiments described in Chapter 4.

Chapter 5 further explored the idea of using the local search for a better exploration of the problem domain and has introduced a greedy MLGLS as another modification of the algorithm allowing to obtain results better than the ones from the GA with an indirect representation. The best results were obtained when a greedy MLGLS was combined with a heuristic initialisation procedure, also examined in Chapter 5. The resulting algorithm is highly efficient and fast optimisation technique suitable for a problem with a large search space. The new algorithm broadens the range of GA-solvable problems by performing a successful search in a large problem domain.

Chapter 6 presents a summary of the candidate's research and suggests some further developments.

The thesis presents the results of multiple experiments conducted during the research in order to evaluate various modifications of the proposed algorithm. The candidate hopes that the thesis will be of interest to the readers that are interested in GAs and their applications to optimisation problems, particularly, scheduling.

CHAPTER 1

Overview of genetic algorithms

1.0 Introduction

This chapter presents an overview of genetic algorithms (GAs) as an optimisation method. After an outline of GAs and their place among other optimisation techniques, an example of a function optimisation by a traditional GA is given in Section 1.2 to illustrate a few basic concepts. The theoretical foundations of GAs, including the Schema Theorem are then discussed in the third section. A number of topics covering the implementation of GAs are presented in Section 1.4, including representational issues as well as variations in GA parameters and operators. Some of these topics are discussed in more detail in Chapters 2 and 3 with the emphasis being on specific aspects of their application as they apply to the case study. The chapter concludes with a short review of some specific techniques in practical GA applications, such as parallel and hybrid GAs, which have been chosen from the multitude of GA techniques due to their relevance to the method suggested in this thesis. These specific techniques will be referred to again in Chapters 3 and 4.

1.1 Genetic algorithms as an optimisation method

Optimisation problems are encountered in all areas of science, engineering, economics and management. Traditional optimisation methods were usually either calculus-based, enumerative or random. *Calculus-based* or *derivative-based* methods were developed in the eighteenth and nineteenth centuries and have been explored very thoroughly since they were the only techniques really available before fast computers were invented. Calculus methods utilise the notion of derivatives and extremal points. An optimum could be found either by solving a set of equations obtained by setting the objective function gradient equal to zero or, alternatively, by 'hill-climbing', that is by moving in the function's steepest direction. Unfortunately, calculus methods are successful only on a limited range of problems. They can be excellent for local optimisation, but on a complicated objective function with multiple optima derivative-based methods are prone to converge to a local optimum. In order to explore the entire problem domain

these methods have to restart from different points. Additionally, they rely heavily on the optimisation function being continuous and, preferably, having a derivative.

Enumerative methods use a rather simple idea: in a finite search space an algorithm examines the objective value at every point in the space. While the enumerative algorithms are straightforward and easy to understand, they may fail to solve real-life problems in an acceptable time due to the fact that the search space is too large (Haupt and Haupt, 1998).

Random searches have gained recognition in recent years as being an alternative to the traditional calculus-based techniques for complicated objective functions and/or large search spaces (Haupt and Haupt, 1998). In their pure form, when the search space is randomly sampled, they are not much more effective then the enumerative methods. However, some kind of a randomised process is used in numerous search techniques. GAs and simulated annealing are examples of more recent methods that use random choice as a part of their search process (Michalewicz, 1999).

GAs belong to a group of optimisation techniques known as *evolutionary algorithms* or *evolutionary computation* (Holland, 1975, Back and Schwefel, 1993; De Jong and Spears, 1993). All evolutionary algorithms represent probabilistic search methods based on the principles of natural evolution: selection and survival of the fittest by passing valuable genetic information down to succeeding generations. Evolutionary algorithms mostly differ in the way they represent a problem and in the choice and probability of genetic operators they use (Back and Schwefel, 1993). However, during last decade the differences between various types of evolutionary algorithms became less obvious due to the fact that in practical applications a number of techniques can be combined to the benefit of the resulting algorithm (De Jong and Spears, 1993).

GAs, for example, originally used binary strings for representation, but a larger variety of representation techniques is used nowadays by GA practitioners (Goldberg, 1991; De Jong and Spears, 1993), which prompted suggestions that the term *genetic algorithms* should be replaced by, for example, *evolution programs* (Michalewicz, 1999). In this thesis, we continue to use the term *genetic algorithms*, however, we should note that in recent years GAs absorbed many characteristics from other evolutionary algorithms, and therefore differ from the original definition of GAs (Holland, 1975; De Jong and Spears, 1993; Michalewicz, 1999).

As noted in (Goldberg 1989), GAs are different from traditional (derivative-based, enumerative) optimisation methods in several ways:

- GAs work not with parameters themselves but with some coding of parameters;
- GAs search from a set of points, not one point;
- GAs use probabilistic transition rules, not purely deterministic rules; and
- GAs use the information from the objective function, not derivatives or other supplementary knowledge.

Outline of genetic algorithms

While a formal definition of GAs components is presented in Section 1.3, an initial outline of GAs is given here as an introduction to this technique:

- GAs imitate natural evolution and collective learning process in a population of individuals (Davis, 1991a). Usually each individual represents a potential solution to a given problem in a search space of all possible solutions. When the first population is initiated, each individual is evaluated according to its performance. An evaluating function introduces a measure of quality for each individual, and therefore provides a means for comparison of individuals. Consequently, this measure gives an ability to decide if an individual is better or worse than the other members of the population. Following this evaluation better individuals are given more opportunity to reproduce (Schwefel and Rudolph, 1995).
- Individuals are usually represented by strings called chromosomes which consist of a number of genes, where each gene characterises some feature of a possible solution (Davis, 1991a). Randomised processes, imitating natural recombination and mutation on the gene level then generate offspring. Recombination provides an opportunity to exchange information between two or more individuals, for example, using a crossover operator, which swaps parts of two chromosomes. Mutation, by comparison, is a self-replication of an individual with possible random modifications. Its aim is to introduce variety into the population and prevent fast convergence to a local optimum. In classical GAs, mutation is secondary to the crossover operator and has a relatively small probability (Goldberg, 1989a).

- After recombination and mutation new individuals evolve with features from both parents and are then evaluated in their turn. The second generation is formed, possibly, with some best parents preserved. After a sufficient number of generations a population is obtained with solutions/individuals approaching the desired optimum (Radcliffe, 1997).

1.2 An example of GA optimisation

1.2.1 Representation

To illustrate the basic features of a GA, consider the following example of a function optimisation. Suppose a function $F(x) = x\sin(5x)$ is to be maximised on the interval $0 \le x \le 2.5$. The function has several local maxima and one global maximum equal to 1.58334545 at the point x=1.5957. A potential solution in that case could be described by a chromosome representing a point from the interval $[0, 2.5]$. Traditionally binary coding was used for representing numerical values. In this case a chromosome consisting of N genes $b_1 b_2 \ldots b_N$, where each of the genes is either 1 or 0, would represent a numerical value of $v^0 = \sum_{i=0}^{N-1} 2^i b_{N-i}$. If it should represent a corresponding part of an interval, as in our case, the numerical value is calculated by the following formula:

$$v = \min + v^0(\max - \min)/(2^N - 1) \qquad (1.1)$$

If the number of genes in a binary chromosome is taken to be 8, the search space of the problem would consist of $2^8 = 256$ points, or 256 parts of the given interval $[0, 2.5]$. Such representation allows the interval to be examined with a precision up to two decimal places. For example, a string 01001101 represents a numerical value of $v^0 = 0*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 77$, or in our case, 77^{th} out of 256 part of the interval $[\min, \max]$. The actual objective value of the point in the search space represented by this chromosome is calculated by substituting $\min = 0$ and $\max = 2.5$ into the equation (1.1). When this is done, the objective value is equal 0.755.

Gray coding

A direct binary representation has a disadvantage in that two adjacent values can be represented by strings that have very little in common. For example, in a binary representation with three genes the adjacent integers 3 and 4 are represented by strings 011 and 100 respectively. It is possible to avoid such discrepancy if Gray coding is used instead of binary coding. The algorithms for converting a binary number into Gray code number and back are given as pseudo-codes in Figure 1.1, while precise formulae for the standard binary and Gray decoding are given in Section 1.4.

```
begin Binary-to-Gray /* b=(b1,...,bN) – binary number,
                     /* g=(g1,...,gN) – Gray code number
    g1=b1
    for i=2:N
        gi=XOR(bi, bi-1)
    end
end
```

Figure 1.1 Pseudo-code for converting binary into Gray code

```
begin Gray-to-Binary /* b=(b1,...,bN) – binary number,
                     /* g=(g1,...,gN) – Gray code number
    b1=g1
    for i=2:N
        if gi=1, bi=NOT(bi-1)
        else bi= bi-1
    end
end
```

Figure 1.2 Pseudo-code for converting Gray code into binary

Table 1.1 gives some examples of binary numbers and corresponding Gray codes. As can be seen from Table 1.1, adjacent integer values differ by one entry only if the Gray coding is used.

Table 1.1 Binary and Gray codes for some numbers

Decimal	Binary	Gray code
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

1.2.2 GA implementation

Figure 1.3 gives the pseudo-code for a traditional GA.

```
begin GA
    t=0
    initialise P(t)
    evaluate P(t)
    until (terminating condition) do
        t=t+1
        P'(t-1)=select_for_reproduction(P(t-1))
        P'(t)=recombine_and_mutate(P'(t-1))
        evaluate P'(t)
        P(t)=select_and_replace(P(t-1) and P'(t))
    end
end
```

Figure 1.3 Pseudo-code for a traditional GA

Let us consider the operation of a GA step by step.

Initial population

Suppose the population size is chosen to be 10. Suppose also that by randomly initialising the population, ten chromosomes are built. When they are decoded from Gray code, corresponding numerical values of the potential solutions x_j and the objective function values, $F(x_j)$ are calculated for all $j = 1, 2, \ldots, 10$. This information

is shown in Table 1.2 while Figure1.4 illustrates the initial population as points on the graph of the objective function.

Table 1.2 Chromosomes in the initial population

Chromosome number	Gray code	Numerical value	Objective function value
1	1 1 0 1 0 1 0 0	1.4902	1.3708
2	1 0 0 1 1 0 0 1	2.3333	-1.8273
3	0 0 1 1 1 1 1 0	0.4216	0.3622
4	0 1 1 1 1 1 0 0	0.8529	-0.7689
5	0 0 0 1 1 1 0 0	0.2255	0.2037
6	0 1 0 0 0 1 0 0	1.1765	-0.4590
7	1 0 1 0 1 0 0 1	2.0196	-1.2593
8	1 0 1 0 0 0 1 1	1.9020	-0.1615
9	1 1 1 1 1 1 1 1	1.6667	1.4788
10	0 0 0 1 1 0 1 0	0.1863	0.1495

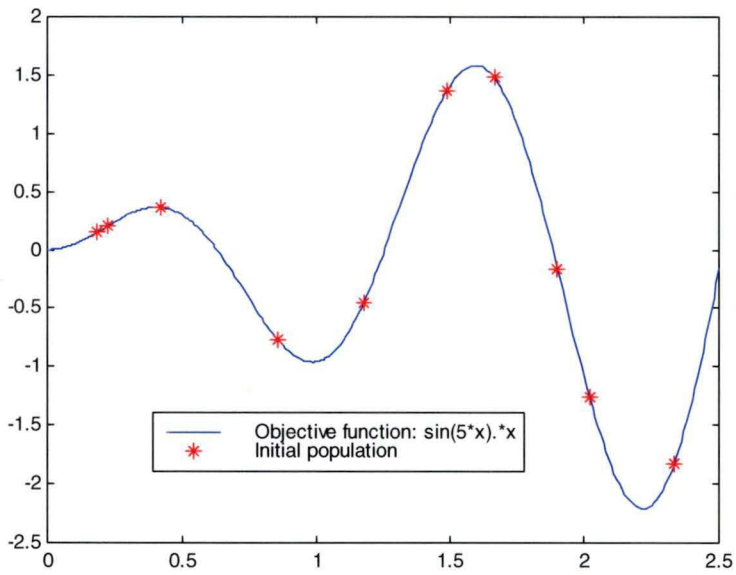

Figure 1.4 Initial population

Elitism and selection

The next step in a GA is to select some individuals for reproduction. One of the usual ways to improve GA performance is to preserve some parent chromosomes for the next population. This is called elitism and in our example one best individual is preserved in

each generation. It means that the selection mechanism should choose 9 out of 10 individuals for reproduction while the best individual is retained as it is.

In this example one of the popular selection mechanisms, the roulette wheel selection (RWS) is used. The RWS mechanism was suggested by Holland (Holland 1975) and its idea is quite simple: each chromosome is assigned a segment of a roulette wheel proportional to its fitness. The wheel is then rotated as many times as needed for the specified number of individuals to be selected. It is obvious that some individuals can be selected many times while others will not be selected at all.

RWS provides good individuals a proportionally better chance to reproduce. Before the selection takes place, the fitness evaluation of a chromosome should be considered. In some GAs the fitness can be equal to the objective function value. In the case of RWS however the fitness values should be non-negative, yet in this example the objective function $F(x)$ is negative in some points of the search space. Therefore some additional measure is needed to transform the objective values into fitness values suitable for RWS selection, for example, subtracting the minimal objective value from all objective values, if the former is negative. This is not necessarily the best possible way to define fitness, but sufficient for the purposes of this optimisation example. The fitness values of the initial population are given in the Table 1.3.

One of the faults of the above fitness assignment is that the chromosomes with the smallest fitness will not get any portion of the wheel if their fitness is negative or zero and, therefore, they will have no chances to reproduce. This may lead to the loss of potentially valuable genes and a population convergence to a local optimum instead of the global one.

Table 1.3 presents an example of an actual RWS when applied to the initial population shown in Table 1.2. As can be seen in the table, fitness of the chromosomes ranges from 0 to 3.3062. The first step in a RWS algorithm is to compute cumulative fitness for the chromosomes. Next, a random number between 0 and 17.3623, which is the largest cumulative fitness, is generated 9 times (the number of chromosomes needed for replacing 90% of the population) and the chromosome whose cumulative fitness is the first fitness greater than the random number is selected. This can be visualised as a roulette wheel with the segments denoted by cumulative fitness and a marker stopped inside a segment.

The random numbers generated and chromosomes that were selected are given in Table 1.3. As shown in the table, chromosome #1 is selected three times, the best chromosome #9 is selected twice, as well as chromosome #10. Chromosomes # 6 and 7 are selected once each.

Table 1.3 Roulette wheel selection

Chrom. number	Objective function value	Fitness	Cumulative fitness	Random numbers								
				16.426	9.9416	1.6491	2.2311	15.9	14.185	2.8965	8.5874	12.333
1	1.3708	3.1981	3.1981			**1.6491**	**2.2311**			**2.8965**		
2	-1.8273	0	3.1981									
3	0.3622	2.1896	5.3877									
4	-0.7689	1.0584	6.4461									
5	0.2037	2.031	8.4772									
6	-0.459	1.3683	9.8455								**8.5874**	
7	-1.2593	0.568	10.4135		**9.9416**							
8	-0.1615	1.6658	12.0793									
9	1.4788	3.3062	15.3855						**14.185**			**12.333**
10	0.1495	1.9768	17.3623	**16.426**				**15.9**				

Crossover and mutation

After the necessary number of individuals are selected, a new population is formed with the help of crossover and mutation operators. In a crossover, pairs of chromosomes exchange their genes after a randomly selected crossover point. The actual crossover performance is shown in Table 1.4. As chromosome #1 was selected in both the third and fourth times, the crossover does not produce any new individuals in the second pair. Since nine chromosomes were selected, one of them does not have a partner and does not participate in the crossover, although it still has a chance to mutate.

Mutation is another genetic operator which randomly changes the value of a gene. Traditionally it is considered secondary to crossover and is performed with a small mutation rate. As shown in Table 1.4, in the GA example examined in this section, just one gene in chromosome #1 was marked for mutation and changed its value.

Table 1.4 Results of crossover and mutation.

Chromosome number	Crossover point	Initial chromosomes	Crossover result	Mutation result
10	4	**00011010**	**00011001**	00010001
7		10101001	10101**010**	10101010
1	2	**11010100**	**11**010100	11010100
1		11010100	11**010100**	11010100
10	6	**00011010**	**000110**11	00011011
9		11111111	11111**110**	11111110
1	5	**11010100**	**11010**100	11010100
6		01000100	01000**100**	01000100
9		11111111	11111111	11111111

Note: a mutating gene is shown in bold

New population

After a new population (children, offspring, second generation) is formed, it replaces the old population less the one elite parent which is retained. The new population is shown in Table 1.5, and it is obvious that the population has improved already. There are two copies of the best individual so far, chromosomes # 1 and 10 (chromosome #1 was the elite in the original generation), while the number of individuals with a negative objective value has reduced from 5 to 2.

Table 1.5 Population obtained after the first generation

Chromosome number	Gray code	Numerical value	Objective function value
1	11111111	1.6667	1.4788
2	00010001	0.2941	0.2926
3	10101010	2.0000	-1.0880
4	11010100	1.4902	1.3708
5	11010100	1.4902	1.3708
6	00011011	0.1765	0.1363
7	11111110	1.6765	1.4478
8	11010100	1.4902	1.3708
9	01000100	1.1765	-0.4590
10	11111111	1.6667	1.4788

1.2.3 Results

The steps that have been described, i.e. the algorithm, is repeated until some termination criterion is satisfied, for example, the maximum number of generations is reached. For this example the population after 20 generations is shown in Table 1.6 and Figure1.5.

Table 1.6 Population at the end of 20 generations

Chromosome number	Gray code	Numerical value	Objective function value
1	1 1 1 1 0 0 1 0	1.5980	1.5832
2	1 1 1 1 0 0 1 0	1.5980	1.5832
3	1 1 0 1 0 0 1 0	1.5294	1.4968
4	1 1 1 1 0 0 1 1	1.5882	1.5822
5	1 1 1 1 0 0 1 0	1.5980	1.5832
6	1 1 1 1 0 0 1 0	1.5980	1.5832
7	1 1 1 1 0 0 1 0	1.5980	1.5832
8	1 1 1 1 0 0 1 0	1.5980	1.5832
9	1 1 1 1 0 0 1 0	1.5980	1.5832
10	1 1 1 1 0 0 1 0	1.5980	1.5832

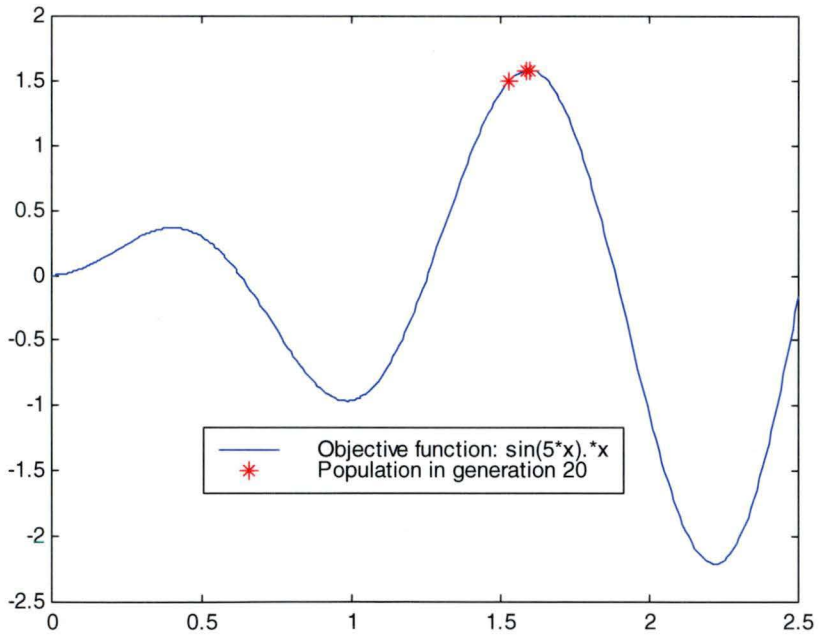

Figure 1.5 GA population at the end of the run (20 generations)

As shown in the Table 1.6 and Figure 1.5, the final population consists of 8 identical individuals with a fitness of 1.5832 which is the closest value to the optimum in our representation. The remaining two individuals are also close enough to the optimum.

1.3 Theoretical foundation of genetic algorithms

1.3.1 Holland's GA

A Holland's GA is a particular type of GA, with the following specific features (Holland, 1975; Schaffer et al, 1998):

- Individuals are represented as bit strings of fixed length N, that is, $a \in \{0,1\}^N$.
- A crossover operator is used for recombination, exchanging sub-strings between randomly chosen individuals. Length and position of the sub-strings are arbitrary but identical for both strings. The crossover probability is usually taken to be from 0.6 to .95.
- The mutation operator randomly changes the bit value with the small probability of 0.01 to 0.001 per bit.
- Assuming a maximisation problem and positive fitness of individuals, a probabilistic selection operator builds the next generation from existing individuals with probabilities proportional to their fitness.

In spite of being a simple algorithm and, in many cases, with only limited possibilities, Holland's GA has received much attention from various researchers and has been thoroughly investigated from the theoretical point of view (Goldberg, 1989a; Grefenstette 1986; Schaffer et al, 1998). As a result, while many GA techniques are based on empirical results, the theoretical foundations of GAs are based on the Holland's GA analysis and, in some cases, have been developed even further to explain more sophisticated types of GAs.

The most common feature of all GAs, and, to some extent, evolutionary algorithms in general is that they are usually described as algorithms processing schemata, also called sub-strings or, from a geometrical point of view, hyperplanes in L-dimensional bit space. The fundamental Schema Theorem (Holland, 1975) analyses exponential growth within a population of relatively short sub-strings that prove to be representative of important features of the solution. These sub-strings are also called building blocks and over generations they tend to accumulate and join with other building blocks to form useful sub-strings of increasing length.

1.3.2 *Schema Theorem*

Holland's fundamental Schema Theorem provides the theoretical base for evolutionary algorithms in general, as well as a particular GA optimisation technique. While it is referred to and described in numerous works produced by various researchers in GAs, one of the best and complete interpretations was given by N.J. Radcliffe in (1997). The following definitions are based on Radcliffe's interpretation.

Definitions

Definition 1.1 (representation, chromosome, gene, allele). Let $\mathcal{S}$ be a *search space*, that is a set of objects over which the search is to be performed. Let $\mathcal{A}_1,\ldots,\mathcal{A}_n$ be arbitrary finite sets with $I = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$. Let g be a function, mapping vectors from I into the search space, that is, $g : I \to \mathcal{S}$. Then the pair (I, g) is called *representation* of $\mathcal{S}$, I is called a *representation space* and g is known as a *growth function.*

The members of the representation space are then called *chromosomes, individuals* or *genotypes.* Each $x \in I$ can be represented as $x = (x_1, x_2, \ldots, x_n) \in \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$ or as a string, $x_1 x_2 \ldots x_n$, where the components x_i are called *genes.* The sets $\mathcal{A}_i$ are usually called *allele sets* or *gene pools* (Radcliffe, 1997; Eshelman, 1997).

Definition 1.2 (schema). Let $I = \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \times \mathcal{A}_n$ be a representation space. For each allele set $\mathcal{A}_i$ an extended set $\mathcal{A}_i^{\#}$ is defined as $\mathcal{A}_i^{\#} = \mathcal{A}_i \cup \{\#\}$, where # is known as a 'wild card' or 'don't care' symbol. Then a *schema* ξ is any member of the set Ξ, defined as

$$\Xi = \mathcal{A}_1^{\#} \times \mathcal{A}_2^{\#} \times \ldots \times \mathcal{A}_n^{\#} \tag{1.2}$$

that is, a chromosome in which some alleles may be replaced with #.

A schema $\xi = (\xi_1, \xi_2, \ldots, \xi_n)$ completely describes a set of chromosomes that have the same values as ξ at all positions i where $\xi_i \neq \#$, the number of such positions being called the order of a schema (Radcliffe, 1997; Schaffer et al, 1998).

Schemata are also known as *hyperplanes* and *similarity templates.* The members of a schema are usually referred to as *instances* (Radcliffe, 1997).

Definition 1.3 (fitness function). Let $\mathcal{S}$ be a search space, $F:\mathcal{S}\to\mathbb{R}_+$ be an objective function and $g:I\to\mathcal{S}$ be a growth function for the representation space I. Then any function $f:I\to\mathbb{R}_+$ will be called a *fitness function,* if the following property holds

$$f(x)=\max_{I}(f)\Leftrightarrow F(g(x))=\operatorname{opt}_{g(I)}(F), \tag{1.3}$$

where opt is minimum, if F to be minimised and maximum if F is to be maximised.

Schema theorem

The following is Holland's Schema Theorem, as formulated by Radcliffe in (1997):

Theorem 1.1 (Schema Theorem). Let ξ be any schema over a representation space I being searched by a traditional GA using fitness-proportional selection, specified recombination and mutation operators and generational update. Let $N_\xi(t)$ denote the number of instances of the schema ξ present in the population at generation t. Then

$$\left\langle N_\xi(t+1)\,|\,N_\xi(t)\right\rangle \geq N_\xi(t)\frac{\hat{f}_\xi(t)}{\bar{f}(t)}[1-D_c(\xi)][1-D_m(\xi)], \tag{1.4}$$

where

- $\langle A\,|\,B\rangle$ denotes the conditional expectation value of A given B;
- $\hat{f}_\xi(t)$ is the *observed fitness* of the schema ξ at generation t, that is, the mean fitness of all chromosomes in the population that are members of ξ;
- $\bar{f}(t)$ is the mean fitness of the entire population at generation t;
- $D_c(\xi)$ and $D_m(\xi)$ are upper bounds on the disruptive effect on schema membership of the chosen crossover and mutation operators, respectively.

Significance and limitations of the Schema Theorem

Most researchers agree that the Schema Theorem, though being simple and easily proved, provides the theoretical foundations for evolutionary algorithms (Goldberg, 1989a; Radcliffe, 1997; Michalewics, 1999). In particular, the theorem can be applied to GAs, if suitable bounds can be estimated for disruptiveness of the operators, which can be easily done in many cases, since most recombination operators that are used in GAs have the property of *respect,* that is, whenever two parents are instances of a particular schema, their offspring obtained via respectful recombination will be instances of the same schema as well (Radcliff, 1991). For example, a specific case of uniform crossover was well investigated (Syswerda, 1989; Spears and De Jong, 1991).

However, it is important to note that the Schema Theorem applies to GAs only if they use fitness-proportional selection (Radcliffe, 1997). Therefore, the theorem cannot be straightforwardly extended to selection methods that depend on the fitness of the offspring, in particular, $(\mu+\lambda)$ and (μ,λ) selection methods used in evolution strategies (Back and Schwefel, 1993). Despite this, multiple practical applications of GAs have demonstrated that a number of different selection and recombination methods work sufficiently well even if they don't comply exactly with the restrictions of the Schema Theorem. It is debated that the most important paradigm that results from the theorem is the *building block hypothesis* which gives some explanation for the success of evolutionary algorithms.

Building block hypothesis

Goldberg formulated a building block hypothesis as: "Short, low-order, and highly fit schemata are sampled, recombined and resampled to form strings of potentially higher fitness" and "...a genetic algorithm seeks near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks" (Goldberg 1989).

This description illustrates the intuitive notion of how GAs work: by combining good features (building blocks) from two or more ancestors, it is often possible to obtain even better offspring. However, it is important to note that it is only a hypothesis, which has not been strictly proved and not even defined strictly enough to be proved.

1.4 Parameters and variations of GAs

1.4.1 Selection

GAs employ two types of selection in two stages of the algorithm: *selection for reproduction* and *selection for replacement.* In each generation some individuals are chosen for reproduction, while later the old population is partly or totally replaced by offspring. In Holland's original GA, individuals were chosen randomly with probability proportional to their performance, that is, better individuals were given more chances to reproduce. Holland's proposal was that only one or two new individuals were created in each generation, and they replaced randomly chosen individuals from the old population. In other modifications of the algorithm, the number of offspring could be equal to the number of parents, replacing the old population completely.

Selection pressure and takeover time

Selection mechanisms are characterised by a parameter called *selection pressure* which relates to the *takeover time* value. This value describes the number of generations needed under pure selection (that is, without recombination and mutation) for an initial individual with the best fitness to fill the entire population (Goldberg, 1989a). If the takeover time of a selection is large it means that the selection pressure is small and vice versa. If a selection pressure is too big, the population loses diversity and converges very fast. On the other hand, a low-pressure selection operator provides mutation and recombination operators with the opportunity to perform a thorough search of the problem's domain (de la Maza and Tidor, 1993; Goldberg and Deb, 1991).

Proportional selection for reproduction

The most popular and widely used selection mechanism for reproduction is *proportional selection* (Back and Hoffmeister, 1991; Bramlette, 1991; Grefenstette, 1997a). Except for *random selection*, when individuals are chosen without any reference to their fitness, selection can be divided into three steps:

- Fitness evaluation according to the objective function;

- Defining an individual's probability to be selected, usually proportional to its fitness; and
- Sampling the population.

Let us consider these three steps in more detail.

Fitness evaluation

The fitness function, as was mentioned in Section 1.3.2, is any function that maps the objective function into non-negative real numbers. If a proportional selection is used, the probability of an individual being chosen for reproduction is a function of its fitness. Although this type of selection is widely used, it may cause a GA to behave very differently when optimising similar functions, like $y = ax^2$ and $y = ax^2 + b$. If the value of b is large compared to the differences in the values of the term ax^2, then the probabilities for selecting the individuals in the population will be very similar and the selection pressure will be very weak.

To avoid this problem, the fitness could be *scaled*, that is, the fitness function is modified to adjust the selection pressure types (Grefenstette, 1986; Goldberg, 1989a). For example, a *linear scaling* function can be used, where the actual chromosome fitness is calculated as

$$f_i' = a \times f_i + b,$$

and parameters a and b are chosen to increase the best fitness compared with the average fitness. This way the fitness of an individual is related to the average fitness and the selection pressure is increased. On the other hand, scaling may lead to dominance of a few good individuals, since if one individual is much better than the rest of the population, it will be selected for reproduction more frequently. As a result, more copies of that individual are obtained in the following generations and a GA will converge prematurely.

One way of avoiding early convergence is to use *ranked selection* (Whitley, 1989), where individuals in the population are assigned a rank according to their fitness and the probability of being selected is a linear function of rank rather than fitness. Ranking in itself can be a linear or nonlinear function of an individual's rank in the population, with the worst individual being assigned zero rank and the best individual having a rank of

M-1 in a population of M. Ranking eliminates the need for fitness scaling and has proven to be an efficient method suitable for many applications (Goldberg and Deb, 1991; Grefenstette, 1997,b).

Selection probabilities

After the fitness values are assigned, a probability distribution should be defined in such a way that the probability of an individual being selected is proportional to the individual's fitness (Grefenstette, 1997,a):

$$P_i = \frac{f(x_i)}{\sum_{i=1}^{N} f(x_i)}$$

Sampling mechanisms for a proportional selection

- The *RWS* technique, discussed in Section 1.2.2 is a popular sampling mechanism used in traditional GAs (Holland, 1975). The probability distribution is used to allocate a segment of the roulette wheel to each individual in the population. The wheel is then spun as many times as the number of parents to be selected, choosing one parent at a time. Such implementation may result in a high variance in the number of children assigned to different individuals, and it is possible that even the best individuals could be overlooked by this sampling and not be selected for reproduction (Grefenstette, 1997,a).
- A *stochastic universal sampling* mechanism (Baker, 1987) differs from the RWS in that it makes a single draw from a uniform distribution and uses it to define the number of offspring assigned to each parent. Continuing the analogy with a roulette wheel, its implementation could be viewed as the roulette wheel being sampled not by one notch at the top, but a given number of notches evenly spaced around the wheel. Then the expected number of offspring of an individual is λP_i. This mechanism exhibits less variance than the repeated calls to the roulette wheel and, as a result, is seen as being more effective (Eshelman, 1997; Grefenstette, 1997,a).

Other types of selection for reproduction

There exist types of selection for reproduction other than purely proportional selection. Some of these selection mechanisms are described below.

- Another popular way of selecting individuals for reproduction is to perform a *tournament selection* (Goldberg and Deb, 1991), when a small subset of the population is chosen randomly and a number of best individuals from this subset are selected. The size of the subset is a parameter which defines the selection pressure. Tournament selection is not affected by scaling or transaction of a fitness function. It is less subject to dominance by a few good individuals and, consequently, premature convergence (Angeline, 1997).

- *Boltzmann selection* thermodynamically controls the selection pressure, using principles from simulated annealing (de la Maza and Tidor, 1993). The fitness function of an individual is defined as following:

$$f(x_i) = \exp(F(x_i)/T),$$

  where $F(x_i)$ is the objective function evaluated for the i^{th} individual and T is a variable temperature parameter (Mahfoud, 1997). Boltzmann selection can be employed to indefinitely prolong the search in order to obtain better final solutions.

- *Disruptive selection* was suggested by Kuo and Hwang in (1993) as the means to explore extreme solutions in the population. Traditional GAs are based on either stabilising or directional selection that tend to eliminate individuals with extreme values or increase the mean value of the population, assuming a maximisation problem (Kuo and Hwang, 1993). To promote diversity in the population and help a GA to find better solutions a disruptive selection was proposed, based on a nonmonotonic fitness function. The fitness is defined as follows:

$$f(x) = \left|F(x) - F(t)\right|$$

  where $F(x)$ is the objective value of the solution x and $F(t)$ is the mean of all solutions in the population at generation t (Kuo and Hwang, 1993). Thus, the solution's fitness increases with its distance from the average of all current solutions, which allows the algorithm to explore the extreme solutions (Hancock, 1997).

Selection strategies for replacement

There are a number of different replacement strategies (Eshelman, 1997). The difference in replacement selection methods could be described as the difference between $(\mu+\lambda)$ and (μ,λ) evolution strategies (Back et al, 1991). If μ parents produce λ offspring, then in $(\mu+\lambda)$ evolution strategies the best μ individuals for the new population are chosen from both generations (Eshelman, 1991). In (μ,λ) evolution strategies μ new individuals are chosen from λ offspring $(\lambda>\mu)$, replacing the parent population completely. Conventionally, the same terminology is used in GA replacement schemes (Back and Hoffmeister, 1991; Michalewicz, 1999).

Generally, the old population is not totally replaced, with at least one copy of the best individual retained for future generations. Such an approach is called *elitist* (De Jong, 1975; Back and Hoffmeister, 1991), and the parameter defining the proportion of the parent population to be replaced is called the *generation gap* (Sarma and de Jong, 1997). If the generation gap is equal to 1, the parent generation is replaced completely. Conversely, if λ in a $(\mu+\lambda)$ strategy is very small, a GA is called a *steady-state* GA. In an extreme case, only one child is created in a generation and the worst parent is replaced (Whitley, 1989).

Another variation in a replacement strategy which helps to sustain diversity of the population and prevent premature convergence is that only individuals that don't have duplicates are included into the new population (Eshelman and Schaffer, 1991; Michalewicz, 1999).

The choice of a reproduction and replacement selection strategy is an important characteristic of a GA (Back et al, 1991). There is no way to predict which selection method would work better on a certain problem, the choice depending on other parameters like the population size and the type of selection used (ranked or scaled, etc). A biased reproduction selection may lead to a premature convergence, while a purely random selection strategy would not produce near optimal results in a relatively short time. Similarly, while a (μ,λ) replacement strategy may lose good individuals forever, it may also lead to an unlikely optimum, which would not be possible if good individuals were retained in the population (Back and Hoffmeister, 1991; Davis, 1991a).

1.4.2 *Representation*

Representation of a potential solution vector to an optimisation problem depends largely on the problem itself. The aim of the optimisation can be anything, from for example, obtaining an optimal configuration for a neuro-fuzzy expert system, modelling stalking behaviour of a predator or evasive tactics of a potential prey, to finding the shortest route connecting all given cities in a travelling salesman problem. Not only do different problems require different representation, but even the same problem could be represented in many ways and the choice of a representation type, together with the other GA parameters, could be crucial for the effectiveness of the search. Therefore, the representation of a solution is one of the most important aspects of GA implementation.

Binary strings vs real-valued vectors

A traditional Holland's GA uses binary strings of fixed length N (Schaffer et al, 1998). That is, the search space I is given as $I=\{0,1\}^N$ and an individual $a \in I$ is a binary vector $a=(a_1,a_2,\ldots,a_N)\in\{0,1\}^N$, which is often referred to as a string $a=a_1a_2\ldots a_N$ where each $a_j \in \{0,1\}$. The mutation operator is then defined as a random inversion of a single a_j variable and the crossover operator exchanges parts of two vectors to produce offspring.

Binary representation is well suited to problems such as a maximum-independent-set problem in graphs (Back and Khuri, 1994), set covering problem (Beasley, 1997), and other so-called pseudo-Boolean optimisation problems, which can be described as $F:\{0,1\}^N \rightarrow \mathbb{R}$. However, traditional Holland's research targeted the problem of a continuous parameter function optimisation, where $F:\mathbb{R}^n \rightarrow \mathbb{R}$, and since then a large number of GA scientists have investigated the same type of problem (De Jong, 1975; Goldberg, 1989a; Davis, 1991a; Michalewicz, 1999). The example of a function optimisation discussed in Section 1.2 has also used binary representation for a continuous function.

When a binary representation is used, it is necessary to define the mechanisms of *encoding* and *decoding* between the two different spaces, $\{0,1\}^N$ and $\mathbb{R}^n$. Usually that would mean restricting the search in whole $\mathbb{R}^n$ to the search in finite intervals $[\min_i, \max_i]$ for each parameter $x_i \in \mathbb{R}$. In that case, the binary vector is divided into n segments of length l_i, such that $N = \sum_{i=1}^{n} l_i$, and then a parameter x_i will be represented by a sub-string of length l_i, $\left(a_{i^0+1}, \ldots, a_{i^0+l_i}\right)$, where $i^0 = \sum_{j=1}^{i-1} l_j$. The *standard binary decoding* function $\Gamma^i : \{0,1\}^{l_i} \to [\min_i, \max_i]$, according to Back (1996), is

$$\Gamma^i\left(a_1, \ldots, a_{l_i}\right) = \min_i + \frac{\max_i - \min_i}{2^{l_i} - 1}\left(\sum_{j=0}^{l_i-1} 2^j a_{l_i-j}\right) \tag{1.5}$$

Another way of decoding is by using the Gray code interpretation of binary strings, which often proves to be more effective since it maps a Euclidean neighbourhood into a Hamming neighbourhood. That becomes possible due to the representation of adjacent integers by bit strings of Hamming distance one, or, in other words, the strings representing the adjacent integers differ only by one entry (Schaffer et al, 1989; Back, 1993). For the *Gray decoding* function Eq.(1.5) is changed into

$$\Gamma^i\left(a_1, \ldots, a_{l_i}\right) = \min_i + \frac{\max_i - \min_i}{2^{l_i} - 1}\left(\sum_{j=0}^{l_i-1} 2^j \left(\otimes_{k=1}^{l_i-j} a_{i^0+k}\right)\right), \tag{1.6}$$

where $\otimes$ is addition modulo two (Back, 1997).

One of the shortcomings of binary representation is that it does not represent the entire search space, with the part it does describe only having a certain degree of precision. Despite this limitation, function optimisation remains one of the traditional areas of binary representation usage. Another area where binary representation could be successfully used is in some types of sequencing problems, such as *job shop scheduling problems* (JSSP) (Nakano and Yamada, 1991; Burke and Smith, 1997). However the majority of practical applications for JSSP use real-value representation.

The main reason why a binary representation produces successful results in some problems is due to the Holland's interpretation of a GA as a schemata processing algorithm which is most successful if the alphabet's cardinality is minimal. Goldberg

generalised this into *the principle of minimal alphabets* (Goldberg, 1989a), arguing that the requirement for binary alphabets can be omitted. This principle has been utilised in multiple practical applications (Davis, 1991a; Michalewicz, 1999) that used *real-valued* vectors as representation for optimisation problems.

There is no clear theoretical or even empirical justification that indicates that binary representation is the best or is suitable for any problem other than the traditional pseudo-Boolean type (Davis, 1991a; Tate and Smith, 1993; Fogel, 1997b). In fact, it has been suggested that for real-valued optimisation problems, *floating-point* representation proves to be more successful than the traditional binary (Michalewicz, 1999). In practical applications it has become obvious that the potential solution implementation should reflect at least some problem specific knowledge, and in some real life problems with a large number of variables it is crucial to incorporate as much additional knowledge as possible in order to get satisfactory results.

Permutations and other representations

One of the popular areas of GA applications is combinatorial optimisation problems such as the *travelling salesman problem* (TSP) and JSSP, in which a potential solution often can be represented as a *permutation* of all cities to be visited or jobs to be performed. However, in a TSP, a permutation is used to describe a cycle, therefore, a number of different individuals will represent the same solution for the problem. Moreover, such representation does not allow the use of traditional mutation and crossover operators. Many researchers have successfully used permutations as GA representation in various combinatorial problems. Goldberg and Lingle (1985) introduced the notion of *ordering schemata,* or *o-schemata* and a *partially mapped crossover* (PMX) operator.

In some optimisation problems a potential solution may contain different types of variables. For example, in many engineering problems some parameters could be integer while others could be of real-value. In that case two types of crossover operators should be defined to deal with the corresponding variables. Thus, a genetic operator becomes a collection of suitable techniques rather than a single operator.

Adjacency matrix representation for a TSP was suggested in (Homaifar et al, 1993). An individual is described as a binary matrix n by n, where n is the number of cities. It has 1 in an entry (i, j) if there is a link between the cities i and j, and 0 otherwise. The matrix crossover is defined as an extension of a conventional one-point or two-point crossover.

1.4.3 Mutation

All GAs find an optimal or near optimal solution by searching the problem space and producing variations in a given population of individuals. The mechanisms for making variations are mutation and recombination. Mutation is applied to a single parent individual, while recombination operates on two or more parents. Recombination represents a powerful exploration capability of GAs, producing new individuals by exchanging genetic information between two parents. By contrast, mutation creates new individuals by randomly modifying existing ones, thus increasing the variety in the population (Davis, 1991a).

Mutation can be defined as a transformation, where small random changes are made in the representation of an existing individual. If a binary representation is used, mutation simply 'flips' binary bits at random. As described by Goldberg (1989, p14), "...Mutation is needed because, even though reproduction and crossover effectively search and recombine extant notions, occasionally they may become overzealous and lose some potentially useful genetic material [...]. In artificial genetic systems, the mutation operator protects against such an irrecoverable loss. [...] By itself, mutation is a random walk through the string space. When used sparingly with reproduction and crossover, it is an insurance policy against premature loss of important notions."

Mutation rate

Traditionally the *mutation rate* is taken to be rather small, on the order of one mutation per thousand bit transfers (Goldberg, 1989a). This is certainly true for simple string encoding with low cardinality alphabets. However, many real life problems require complicated encoding, and in such cases a significantly higher mutation rate could be

more effective (Tate and Smith, 1993). There has been a considerable amount of research into the importance of mutation in GAs and defining an optimal mutation rate (Back, 1993; Holland, 1975; Goldberg, 1989a; Grefenstette, 1986; Schaffer et al, 1998). In GAs, the mutation rate is usually taken to be relatively small, being no more than one over the string length. In certain problems a kind of dynamic control could be beneficial, and variation of the mutation rate over the generations may accelerate optimisation (Back, 1992; Fogarty, 1989; Davis, 1989; Hesser and Manner, 1991).

Mutation in binary strings

In a canonical Holland's GA, mutation operates on binary strings $a=(a_1,a_2,\ldots,a_N)\in\{0,1\}^N$ of fixed length N (Back, 1997). Mutation consists of two steps:

- Randomly determine positions $i_1,i_2,\ldots,i_k$, $i\in\{1,\ldots,N\}$ to undergo mutation, where each position is selected with probability p_m,
- Form a new string with the values at positions $i_1,i_2,\ldots,i_k$ calculated as following:

$$a_i'=\begin{cases}a_i & \text{if } u>p_m,\\ 1-a_i & \text{if } u\le p_m,\end{cases}$$

where $u\in[0,1)$ is a uniform random variable sampled for each $i\in\{1,\ldots,N\}$ (Back et al, 1997, a).

Mutation in real-valued vectors

If a chromosome is represented as a string of real integers $a=(a_1,a_2,\ldots,a_N)$ with genes taking values from finite sets $\mathcal{A}_i$, mutation can be defined as a random changing of an arbitrarily selected gene's value for another one from the same set $\mathcal{A}_i$, which is often called a 'gene pool' in that situation. This is only a generalisation of the above discussed case of binary vectors and is often called *poolwise* mutation (Back et al, 1997a; Eshelman, 1997).

However, real valued vectors may also contain continuous parameters represented as floating point numbers. In this case mutation should be modified to make sure that its outcome is within the problem domain. For example, a *non-uniform mutation* is introduced to fine tune the algorithm (Michalewicz 1999). If a chromosome $a=(a_1,a_2,\ldots,a_N)$ is built from elements $a_k \in [l_k,u_k]$ and an element is selected for mutation, its new value is determined as

$$a_k' = \begin{cases} a_k + \Delta(t, u_k - a_k), & \text{if a random digit is 0,} \\ a_k - \Delta(t, a_k - l_k), & \text{if a random digit is 1} \end{cases}$$

where the function $\Delta(t, y)$ returns a value from the range $[0, y]$ such that the value decreases while t, the number of the current generation, is increasing. As a result, the operator searches uniformly when t is small, that is, in the beginning of the run. The operator searches locally when t increases (Michalewicz, 1999).

Mutation in permutations

If a problem's domain is represented as a set of permutations, it requires specific operators that produce permutations when applied to the individuals of the population. There are several mutation operators that are specifically designed for this purpose, and many of them are related to neighbourhood search operators.

There is a number of widely used mutation operators suitable to deal with permutations:

- *2-opt operator* (Lin and Kernighan, 1973) selects two points in a permutation and reverses the segment between the points. A variant of this operator reverses more than two segments at a time.
- *Insert operator*, which selects an element from a permutation and inserts it into a new position. It is an operator with minimal disruption ability.

Several mutation operators were suggested in (Syswerda, 1991),

- *Position-based mutation*, which selects two elements and moves one in front of the other.

- *Order-based mutation* also selects two elements in a permutation and swaps their positions.
- *Scramble mutation* that randomly reorders a sublist of permutation elements, while all the other elements are left in the same positions.

There may be other mutation operators specifically designed for a particular problem.

1.4.4 Recombination

Recombination, unlike mutation, exploits the idea that if two (or more) individuals perform well, they might exchange valuable information and create a new individual (or a number of them) which may inherit the best feature of all ancestors. As it is unknown a priori what features may be contributing to good performance, the exchange is randomised. Recombination treats the specific features of individuals as building blocks and randomly combines them trying to produce better individuals. In fact, pair-wise recombination is the one feature that distinguishes a GA from other optimisation techniques, like hill-climbing or local search, even if they are population based (Eschelman and Schaffer, 1993).

Binary and integer strings recombination

Holland originally used the basic *crossover* operator, where two arbitrary parents swap all string bits at randomly chosen points. To illustrate the work of a simple *one-point crossover*, consider two individuals x and y, represented by strings of length N. If $k \in \{1,2,\ldots,N-1\}$ is the crossover point, then the crossover operator transforms the parents x and y into two new strings by swapping the parent substrings after the position k:

$$\begin{matrix} x_1 \ldots x_k x_{k+1} \ldots x_N \\ y_1 \ldots y_k y_{k+1} \ldots y_N \end{matrix} \xrightarrow{\text{crossover}} \begin{matrix} x_1 \ldots x_k y_{k+1} \ldots y_N \\ y_1 \ldots y_k x_{k+1} \ldots x_N \end{matrix}$$

It should be noted that although Holland originally applied his definition of a crossover to binary strings, crossover operators work the same way on all linear strings of different alphabet cardinality (Booker in Booker et al, 1997).

Other types of string recombination also exist. These other types include:

- *n-point crossover*, which is a generalisation of a one-point crossover and was first implemented by De Jong (1975). The two-point crossover, when two points are chosen at random and bits of strings between the two points are exchanged is one of the most popular crossover operators, providing an effective search with minimal disruptive effect (Syswerda, 1989; Eschelman et al, 1989; Easton and Mansour, 1993).
- *Segmented crossover* is a variant of a multi-point crossover (Eschelman et al, 1989), but instead of choosing a fixed number of crossover points, it specifies a segment switch rate which defines the probability of segments (that is, segments that are crossed over or segments that are not crossed over), ending at any point in the string. This technique provides a varying number of crossover points.
- *Uniform crossover* is an alternative method where parent individuals exchange randomly chosen bits (Syswerda, 1989). It uses a notion of a crossover mask rather than points and can be beneficial in many cases. If a uniform crossover is used, it can be viewed as a form of adaptive mutation, or convergence-controlled variation (Eshelman, 1997).
- *Shuffle crossover*, is dissimilar to traditional crossover in that it randomly shuffles the bit positions of the two parent strings before crossing them over. After the parts of strings have been exchanged, it unshuffles them. The shuffle crossover was designed "...to eliminate the positional bias of a one-point crossover by having a schema disruption probability that is independent of schema defining length" (Eschelman et al, 1989).
- *Gene pool recombination* builds an individual from genes that are randomly chosen from the gene pool defined by several selected parents (Muhlenbein and Voigt, 1995).

Crossover rate

The crossover is controlled by the *crossover rate,* $p_c \in [0,1]$, which determines the frequency of invoking the operator. It depends on other GA parameters such as the population size, the choice of a selection operator, the mutation rate, etc. Commonly accepted crossover rates are $p_c = 0.6$ (De Jong, 1975), $p_c \in [0.45, 0.95]$ (Grefenstette, 1986), $p_c \in [0.75, 0.95]$ (Schaffer et al, 1989). Some research shows that techniques for dynamically modifying crossover rate could be beneficial (Davis, 1989; Julstrom, 1995).

Recombination operators on real-valued vectors

When real-valued vectors are used as the representation of a problem's search space, a recombination operator can be defined exactly as for the linear strings above, swapping parts of strings after a crossover point. However, other recombination operators could be introduced. Some examples of them are:

- *Intermediate recombination operator*, averaging components of multiple parents. According to Fogel (Booker et al, 1997), a canonical version could be defined as the follows: an offspring $x' = (x'_1, x'_2, \ldots, x'_L)$ is the weighted average of two parents $x_1 = (x_{11}, x_{12}, \ldots, x_{1L})$ and $x_2 = (x_{21}, x_{22}, \ldots, x_{2L})$, that is

$$x'_i = \alpha x_{1i} + (1-\alpha) x_{2i}, \text{ for all } i \in \{1, \ldots, L\}$$

  where $\alpha \in [0,1]$. If $\alpha = 0.5$, the recombination operator produces a simple average of each parameter. Clearly, the operator can be defined to act on more than two parents. In that case it is sometimes called arithmetic crossover (Michalewicz, 1999).

- *Heuristic crossover*, suggested by Wright (1994), uses fitness values to determine direction of search (Michalewicz, 1999). If x_1 and x_2 are the two parents and x_2 is not worse than x_1 and $u \in [0,1]$, then the single offspring is obtained as

$$x' = u(x_2 - x_1) + x_2$$

- *Simplex crossover* of Renders and Bersini (1994), where $k > 2$ parents are selected and the worst of them is determined as, say, x_1. Then the centroid c of the group without x_1 is computed and the resulting vector x' is obtained by moving from c in the direction opposite from the worst parent x_1 (Fogel in Booker at al, 1997), that is,

$$x' = c + (c - x_1).$$

- *Fitness-based scan*, described by Eiben et al (1994), where an offspring of multiple parents is generated from parameters selected from one of the parents with a probability corresponding to the parent's fitness.
- *Diagonal multiparent crossover* also by Eiben et al (1994), which operates like an n-point crossover, except that it takes the first segment from the first parent, the second segment from the second parent and so on.

Recombination in permutations

As was mentioned before, permutations are often used as a search space representation for various combinatorial problems. It is clear that canonical recombination operators would be of little value as they don't necessarily produce a permutation from permutations-parents. For example, consider two individuals-permutations of the set $\{1,\ldots,8\}$ and their offspring produced by a one-point crossover at point k=3:

$$\begin{array}{l} 1\,2\,3\,|\,4\,5\,6\,7\,8 \\ 2\,1\,4\,|\,7\,6\,3\,8\,5 \end{array} \xrightarrow{\text{one-point crossover}} \begin{array}{l} 1\,2\,3\,7\,6\,3\,8\,5 \\ 2\,1\,4\,4\,5\,6\,7\,8 \end{array}.$$

Neither of the offspring is a permutation since the first one duplicates the element 3 while the second offspring duplicates 4.

A number of recombination operators have been designed for various application problems. For example, Davis (1985) and Goldberg and Lingle (1985) defined *order crossover* and *partially mapped crossover* (PMX) respectively to deal with individuals represented as permutations.

According to Whitley (Booker et al, 1997), a variant of Davis' *order crossover* could be explained as following: two permutations are chosen for recombination and one is

called the *cut string* while the other is denoted as the *filler string*. Two crossover points are selected. First, the sublist from the first string between the crossover points is directly copied into the offspring and placed in the same absolute position. Secondly, starting from the second crossover point, the first element in the filler string is found that is not yet in the offspring string. The selected element is moved into position immediately after the second crossover point in the offspring string. The process continues with the next unused elements in the filler string, moving them into the offspring string. When the end of the filler or offspring string is reached, the new elements are looked for or added to the beginning of the string.

The following is an example of Davis' order crossover applied to two permutations with the crossover points after the 3^{rd} and 6^{th} elements. If the parents are given as the following strings,

Parent 1: 1 2 3 | 4 5 6 | 7 8
Parent 2: 2 1 4 | 7 6 3 | 8 5'

then the crossover section from the first parent is 4 5 6 while the filling elements from the second parent in the right order are 8, 2, 1, 7 and 3. The offspring resulting from the order crossover will be as following: 1 7 3 4 5 6 8 2.

Another type of permutation crossover is the *PMX*. It is different from the order crossover in the way it fills the offspring string with the elements from the second parent. In the above example, if a PMX operator was applied to the same parent strings, the crossover section from the first parent, 4, 5 and 6, would be copied into the offspring, as above, resulting in a string with three defined positions, $---|4\ 5\ 6|--$. Then the focus would switch to the elements in the crossover section of the second parent that contains three elements: 7, 6 and 3. 6 has been copied into the offspring already as a part of the crossover section and therefore requires no further action, but the other two elements still have to be mapped into the offspring string.

The idea of the PMX is to try to disrupt the string as little as possible and so 7 and 3 should be placed instead of 4 and 5 respectively, which are in the offspring already. 4 was in the second string in the third place and this is where 7 goes. However, the element 3 cannot replace the element 6 from the second string since 6 is already in the offspring. Therefore 3 replaces the element 5, as it is 5 that filled the position in the offspring that used to belong to 6 in the second parent. The intermediate result is the

string _ _ 7 | 4 5 6 | _ 3 and the rest of the elements from the second parent are copied into their corresponding places, resulting in the string 2 1 7 | 4 5 6 | 8 3.

Other variations of recombination operators include:

- Syswerda (1991) proposed two modifications of the Davis' order crossover. The first modification, the *order-based crossover*, randomly selects several positions in a string and the order in which the elements in these positions appear in one string is imposed onto the other string when constructing an offspring. The second modification, called a *position-based crossover*, imposes the positions of selected elements in one string onto the other string when creating the offspring. While some researchers argue that the two operators are equivalent to each other (Whitley, 1997, C 3.3.3), others note that if the average number of crossover points is significantly less then the string length, the two operators perform differently (Michalewicz 1999).
- *Cycle crossover*, proposed by Oliver et al (1987), partitions two parent permutations into a number of cycles, where a cycle is a subset of elements located at the corresponding subset of positions on both of the parent strings. Recombination selects a few cycles from one parent and the remaining cycles from another.
- *Merge crossover* by Blanton and Wainwright (1993) constructs a single offspring from two parents choosing elements from either of the parents according to a global precedence vector.
- *Edge recombination* was designed specially for the TSP (Whitley et al, 1989). An edge is defined as a link between two cities and is regarded as the basic building block instead of the actual positions of the cities (Homafair et al, 1993). Edge recombination preserves the adjacency between permutation elements, which potentially helps to reduce the cost of a 'tour'. The offspring is built from edges that are present in both parents. This is done with the help of an edge list that contains all the cities connected with the given city in at least one of the parents. When the algorithm starts, the first city is selected randomly from one of the parents, then it is connected to the city from the edge list which has the smallest number of connections, and so forth. The edge recombination proved to be quite successful in a number of applications (Whitley et al, 1989) and has been further developed and improved (Starkweather et al, 1991).

Problem specific recombinations

As mentioned before, there is no clear indication of what parameters would prove most successful in a particular GA problem. However, the growing understanding among GA researchers is that many complex real life problems need to include as much problem-specific knowledge as available into the algorithm. This could be done by choosing a specific representation and/or problem-specific operators. For example, the edge recombination designed for a TSP problem proved to be highly successful due to its emphasis on the edge preservation. Various problem-related crossover operators also have been proposed and implemented (Lee et al, 1993) to enhance GA performance.

In (Cleveland and Smith 1989), the authors suggest several specific recombination operators that are suitable for effective searches in a scheduling flow shop releases problem. Using an ordering representation similar to the one in a TSP, a Goldberg's PMX was used as a possible recombination operator. Besides the Goldberg's PMX, Cleveland and Smith used several other new operators, including:

- *A subtour-swap operator*, which alternatively selects segments from two parents to form a single child.
- *A subtour-chunk "blind" operator*, suggested by Grefenstette in (1987), working similarly to the subtour-swap operator above, but the chunks were placed in the child's chromosome in approximately the same positions as in the parent chromosomes. The chunks are trimmed and slid to the left and right to make them fit.
- *A subtour-replace*, when two similarly located subsequences having the same elements are exchanged to form two offspring. This situation may not be found very often and the operator is recommended as only an auxiliary operator.
- *A weighted chunking*, that uses the domain knowledge to construct better solutions. This is a modification of a subtour chunking operator. When two chunks are fitted into a single child's chromosome, they are moved according to additional information such as production due dates.

Another example of a problem-specific recombination is a *random assorting recombination*, designed for a set recombination problem (Radcliffe and George, 1993).

The recombination "allows controlled sacrifice of respect in pursuit of assortment". The elements for a child are randomly selected from a bag, filled with several copies of elements present in both parents, several 'barred' copies of elements that are present in neither parent, and a copy of each normal element contained in only one parent together with its barred counterpart. This way the offspring will represent an assortment of elements from both parents, but not convey the ultimate respect of recombination.

1.4.5 Review of the current research on GA

Deceptive problems and messy GAs

Deceptive problems are a class of problems that are difficult for a canonical GA to solve. They were first described by Goldberg in (1987). In a simple illustrative example an individual is a string consisting of five-bit segments with the fitness equal to the sum of segment fitnesses, each of them being the total number of ones in a segment. The exception is a segment with five zeros with its fitness being greater than five (Liepins and Vose, 1990). Thus, the problem is called deceptive as the lower-order building blocks may lead away from the global optimum (Homaifar et al, 1991).

One of the most successful strategies to deal with deceptive problems proved to be *messy GAs* (Goldberg et al, 1991; Goldberg et al, 1993). Instead of manipulating building blocks implicitly, they do it explicitly, with a chromosome structure evolving gradually during a run. This is done by introducing a new way of representing chromosomes by strings of variable length, some of them being underspecified or overspecified. That is, some positions can be undefined while others are given conflicting definitions. In the first case the missing value is filled by the value of the corresponding position from a template. In the case of a position being overspecified, that is, given few conflicting values, its first given value is taken to be the current definition and the rest are ignored.

Messy GAs have inner and outer loops. The inner loop performs initialisation of the population consisting of sub-strings of defining length k and evaluation of these sub-strings, discarding the ones of poor quality. Another phase in the inner loop is juxtaposition, when a cut and splice operator, similar to the traditional one-point

crossover applied to strings of various length, modifies existing sub-strings in order to produce better offspring. The outer loop is over levels of defining length, starting with k=1. It continues until some termination criteria are met. At each level the best solution found is used as the competitive template for the next level.

Messy GAs were designed to explicitly implement the building block hypothesis and proved to be an effective method to solve loosely linked deceptive problems, that is, problems with the deceptive sections spread out along the chromosome. Messy GAs proved to be an efficient way to preserve useful links between segments of a chromosome.

Parallel GAs

Parallel GAs are a relatively new concept for evolving a number of sub-populations simultaneously (Pettey and Leuze, 1989; Gordon and Whitley, 1993). The initial motivation for parallel GAs was the potential increase in speed through the assignment to each processor, in a multi-processor system, to evolve a single population. They have also proved to be an efficient method even when implemented on a single processor system (Baluja, 1993). Other reasons to utilise parallel GAs are to maintain diversity in the overall population and to solve problems with multiple optimisation criteria.

There are two varieties of parallel GAs. One method, often referred to as *a distributed GA, island model* or *coarse-grain model*; maintains separate sub-populations that evolve independently. Only occasionally an individual from one population is allowed to migrate into a different sub-population, which ensures information exchange between the sub-populations (Pettey et al, 1987; Goldberg, 1989b; Easton and Mansour, 1993).

The second method, called the *neighbourhood model* or *fine-grain model*, maintains overlapping neighbourhoods. When reproduction takes place, individuals mate and compete with their neighbours within a certain neighbourhood (Muhlenbein, 1989; Manderick and Spiessens, 1989; Spiessens and Manderick, 1991).

A similar approach using a number of sub-populations evolving in parallel is implemented in a *coevolutionary model* (Paredis, 1994), where a population of potential solutions coevolve with a population of constraints. That is, fitter solutions satisfy more constraints while fitter constraints violate more solutions.

Hybrid GAs

Many researchers agree that hybrid algorithms are more efficient than the separate algorithms they consist of (Grefenstette, 1991; Ibaraki, 1997). There is a variety of hybrid GAs which incorporate other optimisation techniques. For example, a number of researchers reported significant improvement in GA performance when combined with a local search method such as hill-climbing (Easton and Mansour, 1993) or neighbourhood search (Ulder et al, 1991; Aarts and Verhoeven, 1997). GAs have been successfully combined with simulated annealing when a general GA frame (or, alternatively, GA with a local search) is enhanced by a selection scheme borrowed from simulated annealing (Mahfoud and Goldberg, 1992; Varanelli et al, 1997).

There is another class of hybrid GA where the GA is a component of a larger system (Ibaraki, 1997). For example, they can control membership functions in a fuzzy system (Karr, 1997) or optimise the design and/or weights in a neural network (Porto, 1997; Goonatilake, 1997). Hybridisation can also be of a reversed type, when fuzzy or neural tools control the GA parameters (Lee and Takagi, 1993).

1.5 Conclusion

This chapter presented a short overview of genetic algorithms (GAs). GAs imitate natural evolution and collective learning process in a population of individuals. As was discussed in this chapter, each individual represents a potential solution to a given problem and is evaluated according to its performance. An evaluating function introduces a measure of quality for each individual, and as a result, better individuals are given a better chance to reproduce. After reproduction new individuals evolve with features from both parents and form a new generation, possibly, with some parents preserved. After a sufficient number of generations, a population evolves with solutions/individuals approaching the desired optimum.

Special attention in this chapter was paid to representational issues in GAs. Individuals in the population are usually represented by strings called chromosomes that consist of a number of genes, where each gene describes some feature of a possible solution. While

originally in GAs individuals were represented by binary strings, GA practitioners nowadays widely use real-value vectors and permutations for representing solutions for various optimisation problems.

Reproduction in GAs is performed with the help of randomised processes imitating natural recombination and mutation on the gene level and producing offspring from the parent population. Recombination provides an opportunity to exchange information between two or more individuals, for example, using a crossover operator, which swaps parts of two chromosomes. Mutation, by comparison, is a self-replication of an individual with possible random changes and its aim is to introduce variety into the population. A number of recombination and mutation operators were discussed in this chapter designed to work on corresponding representation schemes. For example, specific recombination and mutation operators, such as edge recombination and 2-opt mutation have to be used when representing individuals as permutations.

Various evaluation and selection methods were also discussed, including ranking and scaling procedures, roulette-wheel selection, stochastic universal sampling and elitist replacement strategy. The basic GA features were illustrated in a simple example of a function optimisation.

Finally, a short review of several specific techniques in practical GA applications was presented, such as parallel and hybrid GAs. Some of the ideas expressed in these techniques are utilised in the method proposed in this thesis and will be discussed in more detail in following chapters.

CHAPTER 2

Traditional GA approach for maintenance schedule optimisation

2.0 Introduction

The purpose of this chapter is to discuss some of the issues of GA representation for scheduling problems, and to examine the difficulties that a traditional GA may encounter while performing a real-life multi-dimensional optimisation of a maintenance schedule for a power generation system. The case study, including problem specification and data description, is discussed first. Different types of GA representation for scheduling and other combinatorial problems, some of which have already been discussed in the previous chapter, are then reviewed. An indirect representation with two different schedule builders, and a direct representation are then implemented while some specifics of their implementation and GA parameters are clarified. The direct and indirect representations are then compared, with their advantages and disadvantages discussed.

2.1 Scheduling as an optimisation problem

A significant part of research in optimisation is devoted to scheduling and planning tasks that are economically very important, but computationally rather complex. These problems can be found, for example, in production planning, crew timetabling, computer process queuing in an operating system, designing of an autopilot strategy, etc. The traditional travelling salesman problem (TSP) and the job shop scheduling problem (JSS) are examples of scheduling problems that have been intensively studied by a number of researchers (Bagchi et al, 1991; Davis, 1991, a; Syswerda, 1991; Fang et al, 1993; Back et al, 1997,b).

In general, a scheduling problem is a task of allocating a number of jobs over a period of time while optimising specific objectives, for example, cost or completion time; while taking into consideration various constraints such as limited resources or the availability of labour at a particular time. These constraints are often poorly formalised and can contradict each other. Sometimes there is certain problem-specific knowledge that needs to be included into the objective function in order to solve the problem.

Scheduling problems belong to the class of NP-hard combinatorial problems, which means that they cannot be solved by a deterministic algorithm in polynomial time (Garey, 1979). They are computationally difficult and their complexity grows exponentially with the number of parameters considered. A deterministic search of a large space of potential solutions may fail to find a satisfactory solution due to time constraints. On the other hand, a heuristic search is not guaranteed to find an optimal or near optimal solution (Michalewicz, 1999).

In recent years scheduling and planning problems have been the subject of active interest in artificial intelligence research generally and GAs in particular (Husbands et al, 1991, b; Burke et al, 1995). GAs for example, can provide a global search of the problem's domain, and, at the same time, offer a technique that is capable of optimising a poorly structured objective. GAs are thus able to find an optimal or near optimal solution without searching through the entire domain of the problem, (Syswerda, 1991).

However, one of the potential problems encountered by GAs applied to combinatorial problems, such as scheduling, is the enormous search space. As a result, while significant progress has been made in this area of research, only a relatively small portion of it deals with multi-dimensional real life problems searching through a large space (Bruns, 1997).

2.2 Case study: maintenance scheduling in a power system

Power system components undergo planned, regular preventative maintenance in order to minimise the risk of breakdowns. At the same time, any outage in a power system results in some loss in security due to the overall reduction of the generating capacity when power generating units are stopped for maintenance. The purpose of maintenance scheduling in this scenario is to find an acceptable timetable for maintenance outages of the units in the system over a given period of time. The resulting schedule needs to provide a maximum level of efficiency and reliability in the power system (Weedy, 1992; Varga et al, 1999).

Maintenance scheduling involves competition for limited resources and this is often complicated by poorly formalised constraints. In practice the task of scheduling is often

performed manually by a human expert, who uses some variation of an enumerative search technique. This approach cannot guarantee, however, that the optimal or a near optimal solution would be found in an acceptable time, unless the scheduling problem is a rather simple one.

2.2.1 *Problem specification*

Suppose that it is necessary to build a maintenance timetable for a power system over a given period of time (usually a year) which is divided into T equal *time intervals* (weeks, fortnights, months, etc). Let the power system consist of N generating *units* with *capacities* C_j, $j=1,...,N$. The sum of all such capacities is called the *installed generating capacity of the system*, S:

$$S=\sum_{j=1}^{N} C_j \tag{2.1}$$

If the *predicted maximum load* for each interval of the year is known and equal to P_i, for $i=1,\ldots,T$; the *gross reserve* in an interval is calculated as the installed capacity of the system minus the predicted load for the corresponding interval,

$$G_i = S - P_i, \text{ for all } i=1,\ldots,T \tag{2.2}$$

Suppose that each unit j should be maintained once during the entire period considered, with its maintenance requiring a given number of intervals, M_j. Then the system *nett reserve* for an interval i, R_i, is defined as the total installed generating capacity of the system minus the generation loss due to a pre-scheduled outage and minus the maximum load predicted for the interval. In other terms, nett reserve equals gross reserve minus the generation loss due to a pre-scheduled outage,

$$R_i = S - P_i - \sum_{j=J_i} C_j = G_i - \sum_{j=J_i} C_j, \text{ for any } i=1,\ldots,T \tag{2.3}$$

where

T – number of time intervals,

N - number of units,

S - the system's capacity,

C_j - capacity of the unit j, with $\sum_{j=1}^{N} C_j = S$,

P_i - predicted load for the interval i,

J_i - the set of units that are maintained during interval i.

The *level of the system security* is then represented by the minimum of the nett reserve at any given interval. Thus the problem of finding an optimal schedule providing the maximum of the system's security, could be described as following:

$$\text{maximise}\left(\min\{R_i, i=1,...,T\}\right) \tag{2.4}$$

where R_i is the system's nett reserve at interval i, calculated in Eq. (2.3)

Therefore the task of maintenance scheduling in a power system becomes a typical optimisation problem with the objective function being the minimum of the nett reserve of the system over the whole period of time:

$$f^{obj} = \min\{R_i, i=1,...,T\} \tag{2.5}$$

The constraints for this problem can be specified as follows:

(a) Aggregated capacity of the units running at any interval should be no less than the predicted load at that interval, that is

$$S - \sum_{j=J_i}^{N} C_j \geq P_i, \text{ for any } i=1,\ldots,T \tag{2.6}$$

In other words, the nett reserve of the system at any interval should be equal to or greater than zero, $R_i \geq 0$, for any $i=1,\ldots,T$.

(b) Maintenance of any unit starts at the beginning of an interval and finishes at the end of the same or adjacent interval. The maintenance cannot be aborted or finished earlier than scheduled.

2.2.2 *Input data*

The case study described in this research considers the Hungarian Power System. The system consists of 43 power generating units, which are to be scheduled for maintenance during a 52 week period. In this case $N = 43$ and $T = 52$. Data used in the case study is given in Tables 2.1 and 2.2.

Table 2.1. Capacity and maintenance requirements of the units

Unit No.	Capacity (MW)	Maintenance requirements (weeks)	Unit No.	Capacity (MW)	Maintenance requirements (weeks)	Unit No.	Capacity (MW)	Maintenance requirements (weeks)
1	150	3	16	100	4	31	30	3
2	150	3	17	100	4	32	30	3
3	150	3	18	200	4	33	20	2
4	210	4	19	200	4	34	80	1
5	210	4	20	210	4	35	80	1
6	210	4	21	100	5	36	130	3
7	210	4	22	50	4	37	460	8
8	210	4	23	60	4	38	460	11
9	210	4	24	60	4	39	460	6
10	230	8	25	60	4	40	460	4
11	160	2	26	30	7	41	120	1
12	210	4	27	30	7	42	120	1
13	210	4	28	60	7	43	170	1
14	210	4	29	60	7			
15	210	4	30	30	3			

Using Eq. (2.1) it is easy to calculate the installed generating capacity of the system,

$$S = \sum_{j=1}^{43} C_j = 6910 \text{ MW} . \tag{2.7}$$

Table 2.2 contains the predicted maximum load over 52 weeks and the gross reserve of the system at each week, calculated by Eq. (2.2)

Table 2.2. Predicted maximum load and gross reserve of the system

Week	Max Load (MW)	Gross Reserve (MW)	Week	Max Load (MW)	Gross Reserve (MW)	Week	Max Load (MW)	Gross Reserve (MW)	Week	Max Load (MW)	Gross Reserve (MW)
1	5500	1410	**14**	4870	2040	**27**	4490	2420	**40**	4940	1970
2	5650	1260	**15**	4820	2090	**28**	4500	2410	**41**	4980	1930
3	5650	1260	**16**	4850	2060	**29**	4510	2400	**42**	5170	1740
4	5620	1290	**17**	4790	2120	**30**	4520	2390	**43**	5220	1690
5	5600	1310	**18**	4720	2190	**31**	4520	2390	**44**	5370	1540
6	5580	1330	**19**	4660	2250	**32**	4550	2360	**45**	5380	1530
7	5570	1340	**20**	4620	2290	**33**	4560	2350	**46**	5430	1480
8	5520	1390	**21**	4570	2340	**34**	4590	2320	**47**	5460	1450
9	5490	1420	**22**	4540	2370	**35**	4600	2310	**48**	5530	1380
10	5400	1510	**23**	4520	2390	**36**	4670	2240	**49**	5600	1310
11	5330	1580	**24**	4500	2410	**37**	4700	2210	**50**	5590	1320
12	5150	1760	**25**	4520	2390	**38**	4750	2160	**51**	5520	1390
13	4940	1970	**26**	4510	2400	**39**	4820	2090	**52**	5430	1480

2.3 Existing GA techniques for representation of scheduling problems

As was discussed in Chapter 1, selecting the right representation of a potential solution is one of the main issues encountered when a GA is applied to any optimisation problem. The choice of successful representation depends highly on the particular problem being solved. This section continues the overview of existing GA representation techniques, with particular emphasis on scheduling problems. These techniques can be divided into direct or indirect representation, depending on the type of decoding needed for transforming an individual from a GA population into a complete solution, in this case a maintenance schedule (Back et al, 1997; Michalewicz, 1999). If an individual itself represents a schedule or can be turned into one with the help of a simple decoding procedure, such representation is called direct (Bruns, 1993). In an indirect representation, an individual describes a structure completely different from a schedule-solution e.g., a sequence of events (jobs, processes). Because of this an additional transition procedure, usually called a *schedule builder*, is needed to construct a feasible schedule out of the individual using a special algorithm (Bagchi et al, 1991).

2.3.1 *Indirect representation*

A JSS problem, that is, planning a number of jobs (tasks, orders) to be performed on several machines which involves specified time and resources, can be successfully solved using a GA with an indirect representation (Syswerda, 1991; Bagchi et al, 1991). For example, in a scheduling problem in a Test Station laboratory, an individual is represented by a sequence of jobs, such as

$$\{\text{job 1}, \text{job 6}, \text{job 3}, ...\}$$

and a schedule builder transforms this sequence of jobs into a legal schedule using a deterministic set of rules (Syswerda and Palmucci, 1991). According to these rules, the schedule builder takes the first job from the sequence and places it in into the first available time slot. A preference vector that is computed beforehand for each job and depends on the constraints of the particular problem, determines this allocation. The next job from the sequence is then placed into the fist available position in the schedule, taking into account the already scheduled first job. The process then continues for the entire sequence of jobs. The resulting schedule is always legal and its quality depends only on the order in which the jobs are presented in the sequence (Syswerda and Palmucci, 1991).

Such a representation, using the sequence of jobs, transforms the scheduling problem into a sequencing problem, similar to a TSP. Note that for a TSP representation is direct, since any sequence represents a potential solution. For the JSS problem the GA initialises random sequences of jobs to be presented to the schedule builder. Sequence oriented genetic operators, that is, order-based and position-based mutation and recombination are then employed (Syswerda, 1991). These operators change the order of the jobs given to the schedule builder, thereby performing a thorough search in the space of all possible job permutations. For more information on permutations based operators, see Section 1.4.3 and 1.4.4.

This form of Syswerda's indirect representation is called *domain-independent* (Michalewicz, 1999) and therefore contains no information about the scheduling problem itself (Starkweather et al, 1991). As a result, the GA only performs a search in the space of all jobs permutations, with the problem specific knowledge incorporated in

the schedule builder producing a range of feasible solutions. Problem-specific information is also used to evaluate the individuals within the population.

A different approach to an indirect representation, called *problem-specific* indirect representation, is demonstrated in Bagchi et al (1991), where the influence of a schedule builder is described as introducing "...a deterministic aspect to the search which is otherwise stochastic." In this approach an attempt is made to emphasise the stochastic nature of a GA search, and for this reason problem-specific knowledge is incorporated in the individuals. The authors consider a JSS problem, similar to the one that is given in Syswerda (1991), and compare the problem-independent representation suggested by Syswerda with two problem-specific representation versions. In the first version the chromosomes-sequences consist not only of the job numbers but also process plans (Bagchi et al, 1991). Thus, a chromosome has the following structure:

$$\{\text{job 1 / plan A, job 6 / plan C, job 3 / plan B, ...}\}$$

As a result, the genetic search is performed among these process plans, as well as the jobs permutations, while the schedule builder assigns resources to the jobs according to their associated process plans. In a second, expanded version of a problem-specific indirect representation suggested in (Bagchi et al, 1991), the resources (machines to be used in a plan) are also included in the representation. For example, suppose that process plan A for job No.1 is composed of operations *op1A1*, *op1A2* and *op1A3*; and that each of these operations can be performed on the following machines: m1 or m2 for *op1A1*; m3 or m4 for *op1A2*; and m2 or m3 for *op1A3*. Suppose also that process plan C for job No.6 consists of operations *op6C1* and *op6C2*, which can be performed correspondingly on machines m3, m4 or m5, and m2 or m3. Then an individual can be represented in the following way:

$$\{\text{job 1: (op1A1/m1)(op1A2/m3)(op1A3/m3) , job 6:(op6C1/m5)(op6C2/m2), ...}\}$$

In this case the GA does the search through the problem domain, while the schedule builder is only enforcing some of the problem-specific constraints such as the machines' setup times (Bagchi et al, 1991).

In both Bagchi's cases, problem-specific representation genetic operators are modified so that they change string ordering, as well as some problem-specific information, such as the process plans (Bagchi et al, 1991). For example, along with a PMX (see Section

1.4.4), another crossover operator is employed that selects sub-strings in two chromosomes and exchanges process plans plus, in the expanded version, resources between two chromosomes. Mutation is also represented by two operators. The first one is an order based mutation, which swaps elements in two selected positions, while the second mutation operator is problem-specific and randomly changes a process plan. In the expanded version, the second mutation also changes a set of resources in a selected position (Bagchi et al, 1991).

Bagchi's versions of problem-specific indirect representation performed better than a problem-independent representation due to the restricted search space of the latter (Bagchi et al, 1991). While a GA with Syswerda's representation converged to a local optimum after about 100 generations, the problem-specific representation provided an effective search through up to 1000 generations. It also resulted in a better fitness: 64% and 70% in the first and expanded versions compared to only 54% obtained by a problem-independent representation (Bagchi et al, 1991). The authors drew the conclusion that all information relevant to a problem could, and should be represented in an individual.

Other examples of indirect representation of scheduling problems are the following:

- Davis (1985), suggested a representation of a JSS problem as *time-dependent preference lists*, which describe a sequence of jobs presented to a machine, including the machine's starting time, as well as 'wait' and 'idle' elements. An individual would have been represented as:

$$\{(10, \text{job1}, \text{job6}, \text{'wait'}, \text{'idle'}), (60, \text{job3}, \text{job5}, \text{'wait'}, \text{'idle'}), \ldots\},$$

  which would mean that at time 10 a machine searches for a part of job1 that is supposed to be done on this particular machine. If this search is unsuccessful the machine proceeds to the next job in the list, job6, or waits until a part of job6 can be performed. The algorithm uses three problem-specific operators (Michalewisz, 1999): a scramble operator that rearranges the members of a preference list; a crossover operator, exchanging preference lists between selected machines; and a run-idle operator, which inserts the 'idle' element as the second member of the preference list if a machine has been waiting for more than a specified amount of time. The algorithm was successfully tried on a simplified problem (Davis, 1985).

- A bit string representation for a JSS problem where each bit determines the *preference order* of two jobs to be performed on a particular machine (Nakano and Yamada, 1991). Conventional crossover was used and a procedure for repairing illegal schedules was suggested. If a schedule obtained either in the initial population or as a result of genetic operators was illegal, it was replaced by a legal schedule that resembled the illegal as much as possible.
- Cleveland and Smith (1989) investigated a problem of scheduling flow shop releases. An indirect representation was used with individuals describing sequences of jobs, which are placed into a schedule by a deterministic schedule builder. Recombination was performed by one of the following: PMX, subtour-swap, subtour-replace or subtour-chunking operators. A modification of subtour-chunking operator, called weighted chunking operator, was designed to incorporate some problem-specific knowledge, such as job due dates. For more information on these operators see Section 1.4.4. When compared with a direct representation, this type of representation produced better results (Cleveland and Smith, 1989).

In general, researchers agree that a representation incorporating more problem-specific knowledge performs better than a domain-independent one, with the exception of some small scale problems (Cleveland and Smith, 1989; Bagchi et al, 1991; Michalewicz, 1999). Usually, the extent of genetic search has been shown to depend significantly on the amount of knowledge in the representation of individuals, the design of a schedule builder, and specifically constructed genetic operators and evaluation procedures.

2.3.2 *Direct representation*

In a GA with direct representation, an individual describes a complete schedule, timetable, or production plan. This type of representation often requires some specially designed operators in order to preserve the solution's structure and utilise the available problem-specific knowledge (Bruns, 1997).

For example, direct representation is used in a GA application for routing and scheduling trains in a rail network (Gabbert et al, 1991). A chromosome consists of genes corresponding to routes taken by 'blocks', that is, groups of rail cars. The gene

value is an index into a master list of routes stored by the algorithm. This way each chromosome directly represents a potential schedule, which is obtained by simply substituting the genes of corresponding routes. The algorithm uses a uniform crossover operator (see Section 1.4.4) and two types of mutation, one of which is a common poolwise operator which assigns a new route from the master list to a block. The second mutation operator is specifically design to construct a new route with the help of a local optimiser, assign the route to the selected block, and then add the route to the master list (Gabbert et al, 1991).

More complex direct representation is needed for a JSS problem similar to the one described in Syswerda (1991) and Bagchi et al (1991). If a complete production schedule is used as a chromosome, all information relevant to the problem is included in the chromosome (Bruns, 1993). Thus, the process plan for each job, corresponding operations and machines, plus start and finish times of each machine, are all included into the representation. Thus a chromosome may be represented in the following way:

$$\left\{\begin{pmatrix}\text{op1A1/m1}\\ \text{9 to 10}\end{pmatrix}\begin{pmatrix}\text{op1A2/m3}\\ \text{11 to 13}\end{pmatrix}\begin{pmatrix}\text{op1A3/m3}\\ \text{15 to 18}\end{pmatrix},\ldots,\begin{pmatrix}\text{op6C1/m5}\\ \text{8 to 11}\end{pmatrix}\begin{pmatrix}\text{op6C1/m2}\\ \text{12 to 14}\end{pmatrix},\ldots\right\}$$

Note that the order of the jobs in a chromosome is no longer important, since the representation makes any transformation procedures or schedule builders redundant. Thus, the search is performed solely by genetic operators, which have been modified to produce legal offspring (Bruns, 1993). For example, a crossover combines two parents by selecting a number of non-delayed jobs from one parent, taking the missing jobs from the second parent and inserting them into the offspring schedule. Some of the jobs from the second parent may be delayed if they cannot be scheduled in the same time intervals. Mutation is also specifically designed to be able to alter all components in a chromosome. Three types of mutation operators are employed: changing the process plan for a selected job; replacing a machine used in an operation by another free machine; or shifting an operation into the earliest possible time slot. Bruns conducted a series of experiments with GAs, using both a direct and a problem-independent indirect representation for optimising a production schedule for 53 products with up to 3 process plans, up to 19 operations per plan and up to 12 alternative machines per operation. The direct representation produced better results in terms of the sum of squared 'lateness' of the jobs (Bruns, 1993) and was comparable to the problem-specific indirect representation described in Bagchi et al (1991).

Other examples of direct representation of a scheduling problem are:

- A sparse matrix representation (SMR) used for a JSS problem (Liang and Lewis, 1995). A randomly built matrix is decoded into a feasible schedule by a simple procedure, using the knowledge available in the matrix. The SMR operators used are row exchange crossover and row spreading mutation.
- Most TSPs use direct representation when solved by GAs, with a chromosome describing a sequence of cities to be visited, each being visited separately and only once. As was explained in Chapter 1, traditional crossover and mutation would be highly disruptive and inefficient, so a special edge recombination operator was designed (see Section 1.4.4). The mutation operator also differs from a traditional one, with its purpose being simply to change the sequence order (Syswerda, 1991). This approach was extremely successful for the TSP type problems (Whitley et al, 1989; Starkweather et al, 1991).
- A timetable problem investigated by Colorni et al (1991) uses a direct representation with individuals described by matrices or strings of real-value integers. In a school timetable each row of an individual-matrix corresponds to a teacher and each column to an hour. The set of hours to be taught by a teacher is allocated during initialisation of the population, and genetic operators are specifically defined so that they do not disrupt this set of hours. The crossover is designed to preserve good building blocks. It sorts the rows of two matrices in descending order according to specific problem constraints for each teacher. The crossover then takes a given number of best rows from one matrix, filling the missing rows with the corresponding ones from the other matrix. The algorithm uses two types of mutation. The first exchanges several genes for the same number of non-overlapping genes from the same row, provided some specific problem constraints are satisfied. The second type of mutation is a modification of the previous one, swapping one day with another from the same row (Colorni et al, 1991).

It is hard to predict which representation would perform better on a particular problem. When a problem is relatively simple, a straightforward direct implementation with one- or two-point crossover might be suitable. On the other hand, for a problem with a complex structure, a problem-specific representation can be beneficial, although it would require some problem-specific genetic operators to be designed (Bruns, 1997; Michalewicz, 1999).

2.4 Representation of a problem domain for maintenance scheduling

2.4.1 Maintenance scheduling as a representation problem

Maintenance scheduling problems are similar to JSS problems, since they include time-tabling events (maintenance outages), which are subject to time and resources constraints. Maintenance scheduling can use either a direct representation, when an individual denotes a complete schedule (Burke and Smith, 1997), or an indirect representation, with a schedule builder being used to construct schedules from individuals that are represented by sequences of units.

When an indirect representation approach is considered, maintenance scheduling in a power system is different from a JSS problem in two ways. Firstly, each problem requires different factors to be optimised. When a production job shop problem is considered, the goals are normally to minimise the production time and cost while maximising the profit. The production time and cost depend on the jobs being 'packed' as compactly as possible and the schedule builder is designed accordingly, placing each job in the first available slot (Syswerda and Palmucci, 1991). On the other hand, the maintenance scheduling problem is concerned with sustaining a certain amount of the system reserve during the entire maintenance period. Threfore, the approach based on placing each unit in the first available time interval is unsuitable, since such an approach will result in units being scheduled mostly in the first few intervals. If a schedule builder is designed similarly to the one from the JSS problem, it would need to be modified to preserve a certain amount of nett reserve, denoted as R^0, for example. The units are then placed in the first available slot with this retained reserve parameter in mind. Alternatively, a schedule builder would aim to place each unit into the 'deepest' maintenance slot that provides maximum nett reserve.

The second difference is that in a JSS problem a schedule can be considered legal even if some tasks of low priority are left out (Syswerda, 1991). In fact, an indirect representation assumes that the schedule builder is able to make a feasible schedule every time, and then such schedules are then evaluated according to their effectiveness and the number of tasks being included. In a power system, however, one of the

important requirements is that all units have to be scheduled for maintenance during the given time period. Therefore, a schedule builder has to be able to build legal schedules from any random sequence of all the units or, at least, from a majority of such sequences.

Fitness function for chromosome evaluation

For evaluating chromosomes according to their performance, a fitness function should be defined based on the objective function value. Regardless of representation, each chromosome directly or indirectly describes a maintenance schedule, so the fitness evaluation should correspond to the minimum of the nett reserve provided by the schedule at any time. Thus the fitness function of a chromosome l should be defined as:

$$F_l^0 = f_l^{obj} = \min\{R_{li},\ \text{for } i = 1,\ldots,52\}, \tag{2.8}$$

where R_{li} is the system's nett reserve at the week i, provided by the schedule-chromosome l and is given by Eq.(2.3).

However, if the nett reserve at some week is negative, the schedule violates the problem constraints given in Eq. (2.6) and, therefore, the individual is illegal and should not be selected for reproduction. To penalise an illegal individual its fitness needs to be set to any negative value. Threfore, the previous Eq. (2.8) can be rewritten as:

$$F_l^1 = \begin{cases} f_l^{obj} = \min\{R_{li},\ i = 1,\ldots,52\}, & \text{if } f_l^{obj} \geq 0 \\ -c, & \text{otherwise} \end{cases}, \tag{2.9}$$

where c is a positive constant. This fitness function provides the most comprehensive evaluation of an individual according to its performance: if the schedule violates the problem constraints, the performance is unacceptable, or, alternatively, the fitness of a legal individual is equal to the minimal reserve that the schedule provides.

When performing a selection of individuals for reproduction, a scaling or ranking procedure should take place according to the algorithm requirements (see Section 1.4.1). For example, for a proportional selection the fitness function values should be positive, a condition that the above definition does not always satisfy. In this case the fitness can be defined as the rank of the individual within the population. The greater the objective value of the individual, the higher its rank. Individuals with the same objective value are

assigned the same rank value. The solutions that fail to provide positive nett reserve at any time interval are disregarded by being assigned a rank value of zero:

$$F_l^2 = \begin{cases} rank(f_l^{obj}), \text{ if } f_l^{obj} \geq 0 \\ 0, \text{ otherwise} \end{cases} \tag{2.10}$$

In the experiments conducted in this research, the fitness function is calculated using Eq. (2.9); however, when a proportional selection takes place, the fitness is ranked according to Eq. (2.10).

Retained reserve parameter

As mentioned before, if a schedule builder is employed, it should use a retained reserve parameter, R^0. When this parameter is positive, it puts additional pressure on the schedule builder to preserve at least the specified amount of the nett reserve throughout the given period of maintenance. The fitness evaluation procedure can be modified accordingly, that is, if the schedule builder cannot find a maintenance slot for a unit, the individual is declared illegal and is assigned a negative fitness. The preserved amount can be taken as a particular system requirement (Dillon et al, 1976). Alternatively, the preserved amount can be estimated after some consideration of the problem's data. This second approach has been taken in the next section.

Optimum estimation and the optimisation goal

A potential optimal solution was estimated by using the available problem-specific knowledge. As can be seen in Table 2.2, the predicted load reaches its maximum during the second and third weeks, that is, $P_2 = P_3 = 5650$ MW. It means that the gross reserve of the system is at its lowest at that time, $G_2 = G_3 = 1260$ MW. Therefore, even if no units are scheduled for maintenance during these two weeks, 1260 MW is the absolute maximum of the minimal nett reserve of the system over the whole year. This provides an upper boundary for the possible optimal value of the objective function.

The knowledge of the possible maximum of the objective function allows the purpose of the optimisation problem to be defined more clearly. It can be stated that the

optimisation goal is to find a maintenance schedule that will provide a nett reserve of the system as close as possible to 1260 MW at any given week over the year. If the value of 1260 MW proves unattainable, the next best candidate for the optimal objective value could be identified as 1250 MW, or, if that turns out to be impossible to reach as well, 1240 MW etc. For example, it is known that a schedule giving at least 1220 MW of the nett reserve can be built (Varga et al, 1999). Therefore, the aim of this research is to examine the possibility of building a schedule that provides a minimal nett reserve of 1220 to 1260 MW. The experiments described in the thesis make use of the above estimation and they are graded according to the objective value they are set to achieve.

GA parameters

The parameters for a GA can play an important role in an algorithm's performance. While not all of them were exhaustively tested, a sufficient number of experiments were carried out to find a set of parameters that was suitable to test the proposed algorithms.

A GA is more likely to reach a good solution if there is variety in the population, which only a relatively large number of chromosomes can provide (Davis, 1991; Michalewicz, 1999). On the other hand, the larger the population, the slower a GA works and, therefore, computational limitations have to be taken into account. For the latter reason, the largest population considered here is 300 individuals, unless stated otherwise.

After the chromosome fitness is determined by Eq. (2.9) and (2.10), the individuals are selected for reproduction using a stochastic universal sampling selection, which ensures that the fittest individuals have a better chance to reproduce (Hancock, 1997).

An elitist approach is used to improve GA performance, where a number of the best parent chromosomes is taken into the new generation (Goldberg and Deb, 1991; Sarma and De Jong, 1997). The generation gap, discussed earlier in Section 1.4.1, is taken to be equal to 0.9 in experiments described below, unless stated otherwise. That is, 10% of the population represent the 'elite' and are included in the next generation. To ensure that the population is not overcome by a small group of good solutions, new individuals are included only if they are different from all the other individuals already in the population (Whitley, 1989; Eshelman and Schaffer, 1991). All duplicates are discarded and replaced by randomly built new individuals.

2.4.2 *Indirect representation of maintenance scheduling*

Two models of a schedule builder for an indirect representation

When an indirect representation is used, a schedule builder operates on sequences of units, each unit being placed in a sequence once and only once. It means that a population is represented by a set of permutations of all units, which are scheduled one after another by the schedule builder. The number of all possible permutations of the 43 power generating units in this case study is $43! \approx 6 \times 10^{52}$.

To test an indirect representation for the maintenance scheduling problem two models of schedule builders were designed implementing different approaches, 'deepest first' and 'first available'. Both schedule builders transform an individual from a population into a schedule by assigning a maintenance slot to each unit, depending on the system's current reserve. The current reserve R_i^C in week i is calculated each time after a new unit is scheduled, using a modified Eq.(2.3)

$$R_i^C = S - P_i - \sum_{j \in J_i} C_j = G_i - \sum_{j \in J_i} C_j, \text{ for any } i = 1,\ldots,52 \qquad (2.11)$$

where S is the system's capacity, P_i and G_i are respectively the predicted load and gross reserve of the system at a week i and C_j is the capacity of unit j. The set of indices J_i contains all the numbers of the units scheduled for maintenance at week i. Both schedule builders use an additional parameter, a compulsory retained reserve, R^0.

Figure 2.1 presents the pseudo-code for the first schedule builder, implementing the strategy 'deepest first'. The schedule builder places each unit j in the slot that starts with the interval of the highest current reserve so far, provided that in the following M_j intervals, which are required to complete the unit's maintenance, the current reserve is not less than the unit's capacity C_j plus R^0. The individual's fitness, according to Eq. (2.9) is the minimal nett reserve over the maintenance period, and is calculated by the schedule builder while transforming the sequence of units into a schedule. If some units are not scheduled, that is, the nett reserve falls below R^0 when these units are placed anywhere at all, the individual represents an illegal schedule and the fitness should be

assigned a negative value. However, an indirect representation depends on a schedule builder's ability to produce legal solutions from all or almost all individuals. Therefore, if a unit cannot be scheduled while retaining the given amount of reserve R^0, it is scheduled in the deepest slot available, disregarding the retained reserve parameter. Obviously, in this case the individual's fitness will be less than R^0.

```
begin schedule_builder 'deepest first'
define R^0 % define reserve to be retained
units=random_permutation(N) % a random sequence of N units
calculate for i∈{1,...,52}  reserve(i) = Σ_{j=1}^{N} C_j − P_i  % gross reserve at week i
for all j∈ units
   Index=sort_descending(reserve)
   for r={R^0,0} % first 'hard' loop, then 'soft' loop
        for all i∈ Index
             if i+M_j −1≤52 and   reserve(n)−r≥C_j for n∈{i,...,i+M_j −1},
                  schedule(j)= i  % unit j is stops for maintenance at week i
                  reserve(n) = reserve(n)−C_j  for n∈{i,...,i+M_j −1}
                  fit=min(reserve)
                  break % from i loop
             else
                  fit =−1
             endif
        end % i loop
        if fit≥0, % a place for the unit j was found
             break % from r loop
        endif
   end % r loop
end % j loop
end schedule_builder
```

Figure 2.1 Schedule builder 'deepest first'

The second schedule builder employs the 'first available'strategy, placing each unit in the first available slot, provided that the reserve during the following M_j weeks is not less than the unit's capacity plus R^0. When $R^0 = 0$ this schedule builder is similar to the one often used in JSS problems (Syswerda, 1991). As with the first schedule builder, if no place is found for the unit with the required amount of reserve retained, R^0 is

ignored to ensure that the schedule is legal. The pseudo-code for the 'first available' schedule builder is given in Figure 2.2.

begin schedule_builder 'first available'
define R^0 % define reserve to be retained
units=**random_permutation**(*N*) % a random sequence of *N* units
calculate for $i \in \{1,\ldots,52\}$ $reserve(i) = \sum_{j=1}^{N} C_j - P_i$ % gross reserve at week i
for all $j \in units$
  for $r = \{R^0, 0\}$ % first 'hard' loop, then 'soft' loop
    for all $i \in \{1,2,\ldots,52 - M_j + 1\}$
      if $reserve(n) - r \geq C_j$ for $n \in \{i,\ldots,i + M_j - 1\}$,
        schedule(j)= i % unit *j* is stops for maintenance at week i
        $reserve(n) = reserve(n) - C_j$ for $n \in \{i,\ldots,i + M_j - 1\}$
        fit=**min**(*reserve*)
        break % from i loop
      else
        $fit = -1$
      endif
    end % i loop
    if $fit \geq 0$, % a place for the unit *j* was found
      break % from *r* loop
    endif
  end % *r* loop
end % *j* loop
end schedule_builder_2

Figure 2.2 Schedule builder 'first available'

Initial population with indirect representation

The importance of disregarding the required reserve parameter in order to obtain a legal schedule, even if it is of lesser quality, was demonstrated by a series of experiments presented in Table 2.3. In these experiments the 'strict' schedule builders were assigning the maintenance time to the units as described above. The exception was when a unit could not be scheduled without violating the required reserve constraint, then the fitness was assigned a negative value (-1) and the schedule was declared illegal. The amount of illegal solutions in a randomly initialised population of 100 individuals

was evaluated; and the average and the best fitness in the population were calculated. The experiments were conducted with various retain reserve parameter values for 40 runs each. If no legal solutions were found, the run was declared unsuccessful. Data presented in Table 2.3 is the average of the 40 runs.

Table 2.3 Initial population with 'strict' schedule builders

Retained reserve (R^0)	'Deepest first' schedule builder			'First available' schedule builder		
	Number of successful runs	Percentage of illegal solutions	Best fitness in the population	Number of successful runs	Percentage of illegal solutions	Best fitness in the population
0	40	0	1013.00	40	0	8.50
1000	40	53.70	1065.25	40	0	1001.00
1100	40	84.88	1117.50	40	2.23	1100.25
1200	0	100	-1	40	70.83	1200.00
1210	0	100	-1	40	86.83	1210.00
1220	0	100	-1	31	98.48	945.28
1230	0	100	-1	3	99.93	91.33
1240	0	100	-1	0	100	-1
1250	0	100	-1	0	100	-1
1260	0	100	-1	0	100	-1

As can be seen from Table 2.3, both schedule builders produce a population of legal solutions when the retain reserve parameter is small, however all these solutions are of poor quality. As would be expected, this is especially true in the case of the 'first available' schedule builder, since it does not preserve any nett reserve unless forced to do so by the parameter R^0. When R^0=0, the 'first available' schedule builder only occasionally produces solutions with the objective value of 10. The 'deepest first' approach works slightly better, producing solutions with the best objective value of up to 1070 in some runs, but on average of 40 runs the best fitness was only 1013, as shown in Table 2.3. When R^0 increases the quality of the solutions improves but the percentage of illegal solutions in the population increases dramatically and becomes 100% with R^0=1200 for the 'deepest first' schedule builder and R^0=1240 for the 'first available' approach. When R^0 was 1220, the 'first available' schedule builder failed to produce legal schedules in 9 trials out of 40, while with R^0=1230, it found one or two legal schedules in 3 trials out of 40.

Table 2.4 presents the further experimental results of evaluating two 'soft' schedule builders, which implement the exact algorithms described in Figures 2.1 and 2.2. If a

unit could not be scheduled without violating the required reserve constraints, these constraints are ignored and the schedule builders proceed with constructing a schedule with lesser fitness. In contrast to the 'strict' schedule builders above, both 'soft' schedule builders were able to build legal schedules from randomly initialised sequences of units every time.

Table 2.4 Initial population with 'soft' schedule builders

Retained reserve (R^0)	'Deepest first' schedule builder			'First available' schedule builder		
	Best fitness in the population	Average fitness in the population	Percentage of individuals with fitness >= R^0	Best fitness in the population	Average fitness in the population	Percentage of individuals with fitness >= R^0
0	1005.75	742.17	100	8.25	0.16	100
1000	1063.00	875.43	48.18	1001.00	1000.01	100
1100	1112.25	814.11	13.23	1100.00	1089.89	97.80
1200	1040.00	769.74	0	1200.00	940.29	30.40
1210	1038.50	766.79	0	1210.00	892.54	12.80
1220	1039.25	759.94	0	1203.50	851.71	1.65
1230	1037.75	756.36	0	1077.00	817.11	0.13
1240	1042.00	755.44	0	1030.25	773.39	0
1250	1045.75	753.70	0	1007.75	735.44	0
1260	1040.50	753.37	0	934.75	698.05	0

These experiments demonstrate the role of a schedule builder in an indirect representation. In the first series of experiments the schedule builders were too inflexible to produce legal schedules when the pressure to retain a certain amount of reserve was imposed. When this condition was lifted in the 'soft' schedule builders, these schedule builders were able to construct legal schedules, though not always of good quality. Since the ability to produce legal solutions from random sequences is an important requirement for a schedule builder, the 'soft' versions were employed in the experiments below.

Genetic operators and GA parameters for indirect representation

If an indirect representation is used, the search space becomes a set of all possible permutations of units, and representation-specific genetic operators have to be employed. For example, in the experiments described below the recombination operators were either the order crossover (Davis, 1985) or PMX (Goldberg and Lingle,

1985). The crossover rate was taken to be 1 or 0.9. Several mutation operators, that is, 2-opt mutation, insert mutation and scramble mutation, were used simultaneously with the corresponding rates of 0.001-0.005, 0.01-0.05 and 0.0005-0.001. For more information on these operators see Sections 1.4.3 and 1.4.4.

GA performance with the indirect representation

A number of experiments were conducted in order to evaluate a GA with an indirect representation and the results are presented in Table 2.5. Both schedule builders were tried with the retained reserve parameter, R^0, taking values from 0 to 1260. Each experiment was run 40 times for 300 generations. The 'deepest first' schedule builder was always successful with $R^0<1220$, but failed in 7 runs out of 40 with $R^0=1220$; and in 23 runs out of 40 with $R^0=1230$. The 'deepest first' schedule builder failed to produce schedules with a retained reserve parameter greater than 1230. On the other hand, the 'first available' schedule builder was successful in 37 runs out of 40 with $R^0=1240$, and in all runs with R^0 less than 1240. No schedules with $R^0>1240$ were found.

Table 2.5 GA performance with the indirect representation

Retained reserve (R^0)	'Deepest first ' schedule builder				'First available' schedule builder			
	Generation of the first individual with fitness >=R^0	Best fitness in the population	Average fitness in the population	Number of individuals with fitness >=R^0	Generation of the first individual with fitness >=R^0	Best fitness in the population	Average fitness in the population	Number of individuals with fitness >=R^0
0	1	1125	839.38	300	1	23	8.15	300
1000	1	1135	942.78	194.70	1	1010	1002.33	300
1100	1	1160	956.64	153.00	1	1110	1094.39	294.8
1200	36.00	1210	908 69	67 00	1	1209	1073.41	186.6
1210	57.60	1211	910.01	67.80	1	1211	1037 70	137.5
1220	179.00	1222	910.23	70.20	1	1221	989.62	78.3
1230	247 88	1206.25	913.28	78.40	4.2	1230	962.58	68.6
1240	0	1179.00	893.53	0	97	1237.00	952.99	67 5
1250	0	1154.00	880.38	0	0	1150.50	887.42	0
1260	0	1150.00	880.53	0	0	1030.5	807.28	0

In correspondence with Table 2.4, the 'first available' schedule builder was able to find an individual with a fitness of at least R^0 in the first generation in all runs except for $R^0=1230$. The 'deepest first' schedule builder was much slower to find the first

schedule, for example, it took almost 180 generations to find the first individual with fitness 1220, and almost 250 generations before a fitness of 1230 was achieved. Both schedule builders, when successful, were able to produce a number of schedules with a higher retained reserve than required until R^0=1230.

The experiments described in Table 2.5 were conducted with the generation gap parameter being fixed and, therefore, the elite part of the population was always equal to 10%. Nevertheless, by the end of a successful run, 20 to 100% of the population with either schedule builder contained individuals with fitness no less than R^0. It would also be useful to know how fast the algorithm could find a large number of good solutions; that is, the schedules with a retained reserve of at least R^0, if the elite part of the population was allowed to grow, letting all good individuals stay in the population forever. Recall that new individuals are included in the population only if they are different from all these already present.

A group of experiments with a growing elite part of the population were carried out with the retained reserve parameter R^0 being assigned values from 0 to 1240. The results of these experiments are given in Table 2.6. The GA with 300 individuals ran for 300 generations, unless the population was completely filled with good solutions earlier. On the other hand, if there was a significant growth in the number of good solutions at the end of 300 generations, the GA was allowed to run for another 200 generations or until the growth slowed down.

Table 2.6 GA performance with the indirect representation and growing elite

Retained reserve (R^0)	'Deepest first ' schedule builder				'First available' schedule builder			
	Number of generations in a run	Best fitness in the population	Average fitness in the population	Number of individuals with fitness >=R^0	Number of generations in a run	Best fitness in the population	Average fitness in the population	Number of individuals with fitness >=R^0
0	1	1039	742.72	300	1	10	0.15	300
1000	6.21	1079	1014.79	300	1	1004	1000.01	300
1100	13.63	1134	1100.48	300	1.91	1100	1099.39	300
1200	170.72	1201	1196.48	300	9.44	1200	1198.46	300
1210	214.12	1210	1143.85	247.6	18.93	1210	1208 14	300
1220	328.75	1219	921.92	87.52	64.41	1220	1217.79	300
1230	314.19	1206	935.82	151	141.93	1230	1228.07	300
1240	300	1180	889.53	0	200.2	1236.88	1211.73	300

As expected, the GA quickly finds a large number of good schedules when the retained reserve parameter R^0 is small, and takes more generations to fill the population with good schedules when R^0 is large. Still, the 'first available' schedule builder was always successful except for 4 runs with R^0=1240, and needed an average of 200 generations to find 300 individuals with fitness 1240. The 'deepest first' schedule builder was much slower and in 15 successful runs with R^0=1230 found an average of about 150 good individuals in 315 generations.

2.4.3 *Direct representation of maintenance scheduling*

Poolwise representation

When a direct representation is used, each individual in the population should directly represent a possible solution for the optimisation problem, i.e. a maintenance schedule. For a system with a small number of units and just a few intervals in the period of time considered, it is possible to choose a binary representation for the chromosomes. Then every gene in a chromosome describes a complete schedule for the corresponding unit, with '1' standing for the intervals of maintenance and '0' for the intervals when the unit is in operation. An example of such representation, as well as a successful GA implementation is given in (Negnevitsky and Kelareva, 1999; Negnevitsky and Kelareva, 2000). However, when a real life power system is considered, such a representation would result in chromosomes of significant length, and increase the computational time. Moreover, it was shown that for any relatively complex problem, real-valued representation is more effective than binary (Bruns, 1997).

Another approach for representing a schedule is to assign each gene the start of the maintenance outage of the corresponding unit. In the representation implemented in this study, the genes are of integer-value, though a binary coding can be used as well (Burke and Smith, 1997). A chromosome is then built from 43 genes, according to the number of units in the power system, and with the help of a simple decoding procedure completely describes a possible solution to the problem. That is, a chromosome $a=(a_1,a_2,\ldots,a_N)$ is built from genes being taken from a set of allowed values,

$a_j \in \mathcal{A}_j$. The sets $\mathcal{A}_j$, also called gene pools, consist of the intervals' numbers in which the corresponding units can be stopped for maintenance, and are defined as follows:

$$\mathcal{A}_j = \{1,2,\ldots,52-M_j+1\}, \tag{2.12}$$

where M_j is the number of maintenance intervals required for the corresponding unit j for all $j=1,2,\ldots,43$. Gene pools defined by Eq. (2.12) are given in Table A1.1 for all 43 genes representing the power system units. For example, from Table 2.1 it is known that Unit 1 requires three weeks for maintenance and Unit 10 requires eight weeks. Thus a maintenance outage for Unit 1 can start at weeks 1 to 50, while an outage for Unit 10 should start no later than week 45, as shown in Table A1.1. Representation where genes take their values from respective gene pools is often referred to as 'poolwise' representation (Eshelman, 1997).

However, this representation does not guarantee that all chromosomes would represent legal schedules, since too many genes can be assigned the same or similar value. As a result a number of units could be scheduled for outage at the same time. This could make the nett reserve of the system fall below zero for that time, which contradicts the problem constraints given by Eq. (2.6). This is an inevitable trade-off for the simplicity of the direct representation.

Genetic operators for direct representation

When building chromosomes, the value for a gene j is randomly taken from the corresponding gene pool, $\mathcal{A}_j$. Thus, a combination of traditional genetic operators, crossover and mutation, will always produce chromosomes with genes from the corresponding gene pools. Crossover will cut parent chromosomes at a crossover point and exchange the genes after the cut so that each gene will still have an allowed value. In the experiments described in this research one and two-point crossover operators were used. Mutation simply changes a gene's value for another from the corresponding gene pool. This ensures that the resulting chromosomes will consist only of legal genes.

As was stated before, an elitist approach is implemented with 10% of the best individuals from the previous generation preserved. Therefore the crossover rate can be

made equal to 1 in order to maximise the diversity in the population, that is, all individuals chosen for reproduction participate in recombination. The mutation rate is traditionally made relatively small; it is chosen to be 0.05 unless stated otherwise.

Reducing the search space for direct representation

A number of experiments were carried out in order to evaluate the direct representation pertaining to the amount of legal and illegal individuals in a random population. When the gene pools are defined by Eq. (2.12), the percentage of illegal solutions in a randomly initialised population is quite high, almost 30%. This is better than the initial populations produced by the 'strict' schedule builders which had high values of retain reserve parameters (up to 100%, according to Table 2.3), but not as good as the schedules created by the 'soft' schedule builders, which produced only legal schedules by definition (see Section 2.4.2). This draws attention to an important issue: the enormous search space resulting from the use of a direct representation. The number of all possible combinations of genes C is calculated as the product of the gene pool sizes, as follows:

$$C = |\mathcal{A}_1| \times |\mathcal{A}_2| \times \ldots \times |\mathcal{A}_{43}| . \tag{2.13}$$

Namely, there are about 4×10^{72} possible combinations of 43 genes taken from corresponding gene pools, defined by Eq. (2.12).

It is possible to reduce the search space and improve the quality of a random population using the same retained reserve value, R^0, as in indirect representation. However, it plays a different role in direct representation, being a tool in reducing unnecessarily large gene pools. For example, excluding values that cause the system's nett reserve at some period to fall below the chosen threshold could reduce the gene pool for each unit.

As was explained above, a gene's value determines the beginning of the maintenance outage for the corresponding unit. To find new allowed values for each gene, first the gross reserve of the system, defined by Eq (2.2) is modified, by subtracting the threshold value R^0 from it. That is,

$$G_i' = S - P_i - R^0 \tag{2.14}$$

at every interval i. Secondly, only the intervals with a modified gross reserve greater than the unit's capacity, as well as enough subsequent intervals to comply with the maintenance requirements of the unit, are considered as possible corresponding gene values. That means that the definition of a gene pool for a unit j given by (2.12) is changed into the following:

$$\mathcal{A}_j = \{1,\ldots,52-M_j+1\} \cap \left\{ i \in \{1,\ldots,52\} \mid G_i' \geq C_j \right\} \tag{2.15}$$

where C_j is the unit's capacity and M_j is the number of maintenance intervals required. The new, modified gene pools for direct representation with the gene pool threshold equal 1220 and 1260 are given in Tables A1.2 and A1.3. As can be seen from the tables, the gene pools are reduced when the threshold increases, which is especially noticeable when the pools with $R^0=0$ and $R^0=1220$ are compared. For example, when $R^0=1220$ and all unsuitable gene values are excluded, the number of possible combinations is about 2×10^{68}, according to Eq. (2.13).

This procedure complies with the principle of minimal alphabets proposed by Goldberg, which states that it is always advisable to keep the number of allowed values as low as possible (Goldberg, 1989). Such reduction of the search space helps to improve the initial population, as shown in Table 2.7, although there are still illegal individuals in it.

Direct representation performance

A number of experiments were conducted to evaluate the performance of a GA with individuals directly representing maintenance schedules. The population size was 300 while the gene pool threshold was assigned various values, that is, 0, 1220, 1230, 1240, 1250 and 1260 consecutively. The fitness was calculated using Eq. (2.9) and represented the objective value, that is, minimal nett reserve. The results of the experiments are presented in Tables 2.7 and 2.8. Table 2.7 also shows the size of the corresponding search space, which was calculated using Eq. (2.13).

Table 2.7 shows the results of GA optimisation after 300 generations. Reduction of the search space reduced the number of illegal solutions and improved the average fitness in the population at the beginning of a run, with the best individual providing a reserve of

about 100 MW better, and the population's average performance being almost 150 MW better in the experiments with the reduced search space. It should be noted that the gradual increase in the gene pool threshold has very little effect, if any. This means that the search space is still too large and reaching a solution close to the optimum after 300 generations is difficult.

Table 2.7 GA performance with direct representation over 300 generations

Gene pool threshold	Percentage of illegal solutions in the initial population	First generation		Last generation		Search space size
		Best fitness	Average fitness	Best fitness	Average fitness	
0	27.25	785	167.36	1131.30	811.71	4.23E+72
1220	13.55	870	311.13	1150.73	822.06	2.31E+68
1230	14.35	859	309.77	1147.00	820.03	1.88E+68
1240	13.13	872	317.36	1149.50	841.44	7.77E+67
1250	14.53	852	301.35	1151.00	839.22	4.15E+67
1260	14.25	856	297.08	1150.00	818.81	1.87E+67

In the next series of experiments, a GA with various gene pool thresholds was run for 5,000 generations in order to examine its performance in more detail. The results, presented in Table 2.8, include the best and average fitness in the last generation, as well as the generation of the last improvement of the best fitness. As can be seen from the table, there was no significant difference with the results from 300 generations, and the best schedules in all runs were providing the retained nett reserve of only 1190 MW. However, it should be noted that the GA with the reduced search space stoped its improvement on average 500 to 1,000 generations earlier.

Table 2.8 GA performance with direct representation over 5,000 generations

Gene pool threshold	Best fitness	Average fitness	Generation of the last improvement
0	1142 (1080 to 1180)	815.45	2132 (113 to 4965)
1220	1154 (1120 to 1190)	821.037	1725.1 (60 to 4397)
1230	1155 (1120 to 1190)	819.73	1345.2 (108 to 3650)
1240	1153 (1130 to 1170)	816.97	1399.6 (258 to 4633)
1250	1160 (1130 to 1180)	816.107	988.2 (406 to 1920)
1260	1149 (1120 to 1190)	808.298	1044.5 (110 to 3046)

A typical performance graph for the traditional GA with the retained reserve parameter R^0=1230 is given in Figure 2.3, presenting the best and the average fitness (nett reserve) in the population. It shows how very slowly the traditional GA converges.

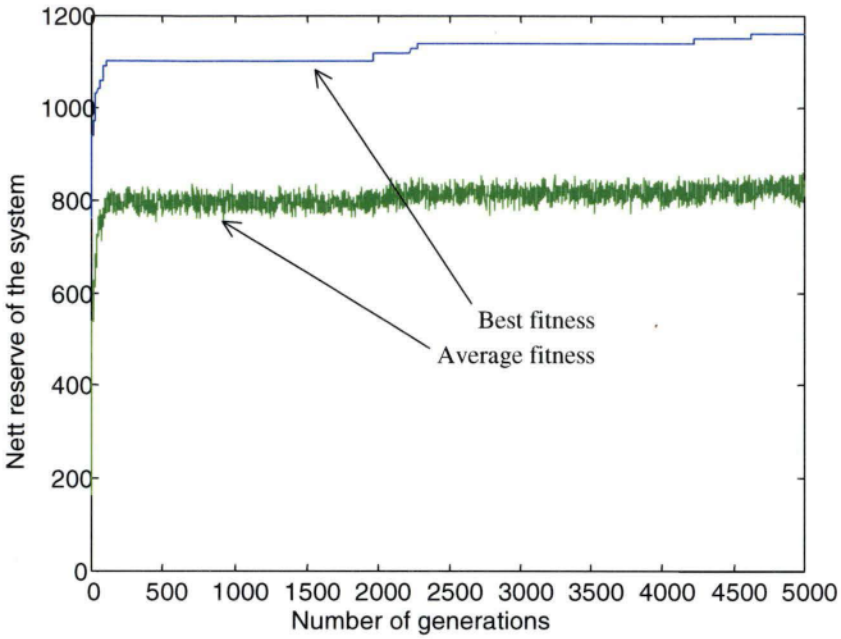

Figure 2.3 Performance of a traditional GA

Direct representation with seeded first generation

It usually helps to improve a GA performance if some problem specific knowledge is incorporated into the algorithm, for example, the best solutions obtained in previous runs are used to seed the initial population (Grefenstette, 1987; Fang et al, 1993; Eshelman, 1997). A series of experiments were carried out using the schedules obtained by a schedule builder with indirect representation to seed the initial population with individuals directly representing schedules. The 'seeds' were schedules that provided reserves of 1220, 1230 and 1240 MW and were used in conjunction with a retained reserve parameter taking values from 1220 to 1260. Table 2.9 presents the results of the experiments when the two seeds were solutions with fitness 1240.

The experiments were run for up to 1,500 generations and, as seen from Table 2.9, the algorithm filled the elite part of the population with individuals having fitness of 1240, but failed to improve the best fitness in the population beyond the seeded value. When

the algorithm was run with 1, 10 and 50 seeds with fitness of 1220, 1230 or 1240 in the initial population, the results were similar.

Table 2.9 Seeded GA performance with direct representation

Gene pool threshold (R^0)	First generation		Last generation		Number of solutions with fitness >=R^0	Number of solutions with fitness >R^0	Number of generations to fill the population
	Best fitness	Average fitness	Best fitness	Average fitness			
1220	1240	317.68	1240	890.89	32.95	32.65	215.70
1230	1240	321.54	1240	888.11	32.70	32.50	302.50
1240	1240	321.05	1240	886.35	32.75	0	505.70
1250	1240	310.48	1240	890.03	0	0	n/a
1260	1240	305.19	1240	888.17	0	0	n/a

When the experiments were run with growing elite, the population was soon filled with individuals having the same fitness as the initial 'seed', or less, if the pool threshold was less than this fitness. The number of generations needed to fill the population is also presented in Table 2.9.

2.4.4 Comparison of indirect and direct representation methods

After examining a GA with different types of representation, the influence of a representation on GA behaviour is discussed in this section.

Performance

As shown in Tables 2.3-2.9, a GA with indirect representation and a schedule builder performs much better than a GA with individuals directly representing schedules. The schedule builders were able to produce better individuals from the very start, making sure that all solutions in the population are legal. When the algorithm was run for a number of generations, it developed schedules of good quality, as was shown in Tables 2.5 and 2.6.

In contrast, a direct representation was unable to find good solutions due to the large scarch space, even when a space reduction procedure was implemented. There was no

guarantee that all solutions in a population would be legal, and the best objective value that the GA was able to reach in all runs was 1190, significantly less than the 1240 reached by a 'first available' schedule builder.

Representation mapping into the problem domain

Still, an indirect representation has a substantial disadvantage since it does not represent all possible schedules, but only the ones that a schedule builder can produce from a sequence of units. Thus, it lacks a very important quality, that is, mapping the representation space into the entire problem's domain (Bruns, 1993), and its search is only as efficient as the schedule builder implemented in the algorithm, which explains, for example, the difference in the performance of the schedule builders examined above. On the other hand, a direct representation includes all possible solutions, and even a large amount of illegal solutions, which is undesirable, but not crucial for a GA performance. Reducing gene pools, as was shown above, can lower the percentage of illegal solutions by half. There also exist various techniques to optimise a GA search in a large space, but it is very important that this space provides a complete representation of the problem domain.

Preserving the similarities

A good representation should obey some structure preserving conditions (Radcliffe, 1991; Back, 1997), that is, solutions with similar qualities should be represented by similar individuals. Both direct and indirect representations meet this requirement to some degree. When a sequence of units is altered by changing the order of two adjacent units, it depends on these particular units and their positions in the sequence, as to whether the resulting schedules will be identical, similar or completely different. The same can be said about direct representation: if a gene's value is shifted, it may lead to a change in the objective value, depending on the unit's characteristics.

It is highly improbable to devise a completely structural representation for a real life problem, and if it were, then there would be no need for a GA approach, since such a problem would be solved more easily by a derivative based method (Back, 1997).

Illegal solutions in a population

Usually a representation maps a population of individuals into the whole search space of a problem, not necessarily just the space of legal solutions. Therefore, minimising the amount of illegal solutions is an important issue in representation. When the large search space of a direct representation is considered, some schedules are inevitably illegal, although adjusting the gene pools can reduce their number. Sometimes, an additional step is taken to ensure that all individuals in the population represent feasible solutions, and for this reason the schedule builders discussed above have been modified to produce legal schedules all the time. An initial, directly represented population can be improved using so-called 'random extended initialisation', where each individual is the best of N randomly built individuals (Bramlette, 1991). Alternatively, a schedule builder can be employed to initialise a population of schedules (Bruns, 1991).

However, even if all individuals are legal, it does not guarantee that the result of a crossover or mutation will always be legal. In such case a repair algorithm can be introduced to generate a legal solution as close as possible to the illegal one (Nakano and Yamada, 1991). Otherwise, a few individuals that do not satisfy the problem's constraints are penalised during the fitness evaluation and not selected for reproduction (Michalewicz, 1999).

Redundancy of representation

An indirect representation has a disadvantage as a result of dealing with the structures that are only a step towards solutions, and not the solutions themselves. That is, two individuals can be transformed into the same schedule, thus, a single solution can be represented by several points in a search space. Such a situation is usually referred to as 'redundancy of representation' and is highly undesirable (Radcliffe, 1991; Faulkenauer, 1994).

For example, if a sequence, starting with the following units

$$\{38, 37, 40, 15, 6, 39, 7, \ldots\}$$

is transformed into a schedule by the 'deepest first' schedule builder with the retained reserve parameter R^0=1220, the units are assigned the corresponding weeks for the beginning of the maintenance outage:

$$\{27, 19, 18, 38, 15, 27, 14,...\},$$

and then the rest of units from the sequence are scheduled. If, however, two units in the sequence, 39 and 7, change places, they will still be scheduled into the same slots, assigning week 14 to unit 7 and week 27 to unit 39. Obviously, the two sequences then represent different individuals in the population but the same schedule. Note also that units 39 and 7 are quite different, with capacities of 460 MW and 210 MW respectively. The maintenance requirements are different too, being 6 and 4 weeks correspondingly.

In the experiments described above, one more parameter, the proportion of individuals in the population representing unique schedules, was also measured. When an indirect representation was used, this proportion was often quite low (25 – 60 %), and a schedule builder transformed a large number of individuals into the same schedule. This was especially true in the experiments with the growing elite part. For example, in one particular case with the 'deepest first' schedule builder and the growing elite proportion of the population, 300 individuals in the final population with the retained reserve of 1220 MW represented only 37 distinct schedules, with one particular schedule having 61 copies, even though all 300 sequences of units were different. When the 'first available' schedule builder with R^0=1240 and growing elite was examined, the average number of different schedules represented by 300 sequences in the last generation was 84, with the actual number ranging from 20 to 218.

By comparison when a direct representation was used and the initial population was seeded with good individuals, all solutions produced by the algorithm were different, since new schedules were introduced into the population only if they were different from all other members of the population.

Note that there is a specific type of similarity of the solutions regardless of representation, due to the fact that some units have the same capacity and maintenance requirement and obviously can be interchanged either in a schedule or in a sequence of units presented to a schedule builder. This type of redundancy is an attribute of the particular problem and has to be tolerated. This attribute will be discussed in more detail in the next chapter.

2.5 Conclusion

In this chapter, a special type of optimisation problem, scheduling, was analysed. A case study, a maintenance scheduling optimisation in a power system, was considered.

Direct and indirect representation of the maintenance scheduling problem was discussed and implemented. In a direct representation GA, an individual describes a complete schedule, that is, genes in a chromosome denote the beginning of a maintenance outage of the corresponding unit. When an indirect representation is used, an individual represents not a schedule itself, but a sequence of units that are scheduled for maintenance by a separate schedule builder. Two models of a schedule builder are proposed in this chapter. These models implement two different methods of scheduling. The first one places each unit in the time slot starting with the interval of the largest nett reserve, while the second assigns each unit for maintenance into the first available time interval.

The results of the experiments with the schedule builders are presented. Both schedule builders are able to find solutions of good quality, the second schedule builder being more successful, and producing a schedule with a nett reserve of 1240 MW. However, the indirect representation was unable to find schedules providing a nett reserve of more than 1240 MW, and the question remains whether it is possible to build schedules that exceed this reserve. The ability of a GA with an indirect representation to answer this question is limited, because the algorithm examines only the restricted search space defined by the design of the schedule builder. Moreover, an indirect representation is highly redundant and many different sequences are transformed into the same schedule. Nevertheless, the indirect representation provided a better estimate of the possible optimum, and the solutions found so far could be used for seeding a GA population when a different search strategy is used.

In contrast, a direct representation provides the possibility of examining the entire domain of the problem, and according to the Holland's Schemata Theorem, given sufficient time, the optimum will be found. The major problem here is the large representational space (up to an order of 10^{72} possible combinations of unit outage starting times instead of an order of 10^{52} permutations of units for an indirect representation) and, as a result, a very slow search. It was demonstrated that a GA with

direct representation failed to find schedules with nett reserve greater than 1190 MW, unless seeded with some good schedules found by a schedule builder.

In conclusion, to examine the entire problem domain more thoroughly, a direct representation should be used although some measures need to be taken to improve GA performance. A method that radically optimises the search is proposed in the next chapter, performing an effective search in a large representation space.

CHAPTER 3

Multi-layered genetic algorithm

3.0 Introduction

In this chapter we continue the examination of the case study. Firstly, a number of search strategies that can be used to enhance GA efficiency are reviewed. These strategies include delta coding, parallel GAs, grouping GAs and multi-chromosome representation. Next, a new approach, a multi-layered genetic algorithm (MLGA), first reported in (Kelareva and Negnevitsky, 2001a), is proposed, and implementation issues, such as chromosome structure, gene pool criteria and termination criteria, are explained. Two examples of unit groupings are presented, and the performance of an MLGA with these types of groupings is discussed. Finally, to enhance the MLGA search technique, a modification of the gene pool criteria is suggested, and its effect on MLGA performance examined.

3.1 Effective search strategies for a large problem domain

As was shown in the previous chapter, GAs may encounter significant difficulties when exploring the large domain of a real life problem with multiple parameters. A direct representation results in a search space too large to perform an effective search, while an indirect representation proves to be more efficient for the case study problem, being able to obtain better results. Yet a GA with an indirect representation does not explore the entire problem domain and, therefore, it is unclear whether the domain has been examined in the best possible way or whether the obtained results represent optimal solutions. Therefore, it would be useful to investigate the direct representation space more thoroughly and to consider the possibility of improving the results obtained in the previous chapter.

There are various techniques to enhance GA performance. New ways of representation, modified genetic operators and various techniques for tuning GA parameters have been extensively investigated by a vast number of researchers (Bagchi et al, 1991; Aizawa and Wah, 1993; Eiben et al, 1994; Falkenauer, 1997). Usually a GA should be tailored to suit a particular problem (Davis, 1991). Some representation and/or operators may work for one problem, but prove unsuccessful for another (Michalewicz, 1999).

Moreover, a GA may work remarkably well on a trial problem with a minimised number of parameters and a small search space, but the same problem with multiple variables taking a large number of values may be impossible to solve with the same algorithm. For example, a maintenance scheduling problem similar to the case study discussed in Chapter 2, but with a small number of units scheduled during 12 time intervals, was successfully solved by a traditional GA, as shown in (Negnevitsky and Kelareva, 1999; Negnevitsky and Kelareva, 2000).

Recently a considerable amount of research has been devoted to GA applications in multi-dimensional problems (Husbands and Mill, 1991; Bruns, 1997). There are a number of effective search methods that follow the idea expressed in the building block hypothesis (Goldberg, 1989), and concentrate their efforts on the formation of building blocks of increasing length, if these building blocks prove to be beneficial.

- One of the examples of such methods are *messy GAs*: "...messy genetic algorithms find and emphasise tightly coded substrings initially, juxtaposing them thereafter to find globally optimal structures" (Goldberg et al, 1991). This has proven to be an efficient way to preserve useful links between the segments of a chromosome. Messy GAs are described in more detail in Section 1.5.1.

- The idea of promoting good building blocks has been also used in a JSSP application with lot sizing and sequencing (Lee et al, 1993). First, a traditional chromosome representation was used, where every gene represented a job type with a small lot size. The algorithm was run for a few generations to enable the GA to find a good clustering of genes, and then clusters of the same type job were coalesced into a single lot. That is, several primitive genes were replaced by a single building block in all chromosomes in future generations, thus reducing the chromosome length. The empirical results of this approach showed a significant improvement in performance (Lee et al, 1993).

- A new way of representing individuals, a *multi-chromosome representation*, was suggested in a problem for pallet loading, where three chromosomes were used to represent all the features of an individual (Juliff, 1993). The genetic operators were applied separately to these chromosomes, that is, parts of an individual were treated differently, depending on their interpretation. This multi-chromosome representation can also be viewed as another example of preserving some chromosome structural characteristics, with each chromosome of an individual acting as a separate building

block of a large size. According to the author, this technique allowed the load builder to use all the information available about the pallet type and ordering and, as a result, to produce more meaningful solutions.

- The idea of gradually building good solutions is used in a *co-evolving GA*, suggested by Husbands and Mill (1991). This algorithm represents a variation on parallel GAs and was implemented on a parallel computer for a generalised JSS problem. In this algorithm a parallel GA is used for simultaneously solving a number of interacting and competing sub-problems and, as a result, a solution to some wider and more complex problem is gradually built. Separate populations, or species, in the co-evolving GA have different chromosome structure, with each species representing a potential solution to a sub-problem. For example, each population can consist of individuals that represent production plans for a particular product to be manufactured (Husbands and Mill, 1991). When fitness is evaluated, an individual's interaction with the members of other populations is taken into account and members of a special population, the Arbitrators, decide possible conflicts over resources. As a result, the "... separate species co-evolve in a shared world..." (Husbands and Mill, 1991). It can be said that this co-evolving GA first defines the structure of a number of building blocks, then finds good representatives of such building blocks and combines them to form a complete solution to a problem.

- A number of grouping problems have been solved using a special type of GAs, called the *grouping GAs*, which also promote the idea of larger building blocks (Falkenauer, 1994 and 1996). Grouping GAs operate not on single objects but on groups of objects. This approach requires a specific representation and specially designed genetic operators, which ensure that offspring inherit favourable features from their parents. The reason for the success of the grouping GA is that rather than trying to combine the building blocks from single genes/objects, they operate on genes that are, in fact, meaningful building blocks representing groups of objects. Grouping GAs have proven to be successful in several applications, for example in assembly line scheduling (Falkenauer, 1997).

These examples demonstrate that rather than letting a GA combine genes in the traditional way, an algorithm lends itself to the formation of meaningful and successful

building blocks. Furthermore, an appropriate representation, often dynamically adapted when the population is evolving, is beneficial (Michalewicz, 1999).

Another tendency that can be noticed in some of the effective GA methods is the search being focused on a promising area of the problem domain instead of performing the search according to traditional GA paradigms.

- An example of such a method is a *delta coding algorithm* (Whitley et al, 1991). This type of algorithm is used for optimising a geometric transformation, and consists of a series of consecutive runs, iteratively improving a sub-optimal solution until a desired precision is obtained, or time constraints are satisfied. The first run in the delta coding algorithm acts as an ordinary GA, and the best individual found is passed to the next run as a partial solution. Subsequent runs form new individuals by encoding each of their parameters as delta value ($\pm\delta$), that is, a distance from the value of the corresponding parameter in the most recent partial solution. This procedure effectively builds a 'delta hypercube' around the partial solution found in the previous run. The hypercube can be enlarged or reduced according to the needs of the algorithm at the start of a new run, focusing the subsequent search into the area that has already been shown to be successful. This way delta coding directs the search to a specified area by preserving the best result found so far, and at the same time has the advantage of promoting the diversity of the population, since new random individuals are introduced at the beginning of every run (Michalewisz, 1999).

In fact, focusing a genetic search in some promising area is widely used, even if not emphasised in other methods. For example, using *'random extended initialisation'*, where each individual is the best of a few randomly built individuals (Bramlette, 1991) or seeding an initial population with good solutions obtained in earlier runs or by other methods is a known way to improve GA performance (Fang et al, 1993; Eshelman, 1997). Recall that in the case study a traditional GA with direct representation performed better when seeded with individuals found by a schedule builder, according to Tables 2.8 and 2.9.

3.2 What is a multi-layered genetic algorithm?

3.2.1 Separability of a problem

As described by Radcliffe: "One of Holland's basic motivations and beliefs was that complex problems are most easily solved by breaking them down into a set of simpler ... subproblems, and this belief is visible ... in his conception of schema analysis" (Radcliffe, 1997).

The search strategies discussed above, particularly the ones that emphasise the building block formation, employ to some degree the idea of dividing a problem into sub-problems, and depend on the partial separability of the problem being solved. This is especially evident in the co-evolving GA, where "the idea is to recast a highly complex problem in terms of the cooperative and simultaneous solutions of a number of simpler interacting sub-problems" (Husbands and Mill, 1991).

If a problem can be decomposed into independent sub-problems, it is called *linear separable* (Michalewicz, 1999). It is obvious that only a few very simple problems would be completely linear separable and, in fact, they most probably would be more easily solved by other optimisation methods (Radcliffe, 1997). GAs do not need complete separability, but the success of GA operators in recombining potentially beneficial building blocks into a suitable solution does depend on a certain degree of separability, which often can be achieved by choosing the right representation for the problem (Davidor, 1990).

In the case study, if a direct representation is used, chromosomes represent maintenance schedules built from genes that denote the beginning of the maintenance periods of the corresponding units. It is clear that the problem is not completely separable, since we cannot divide the units into groups to be scheduled independently from each other and then combine two schedules into one. For example, two groups of units might be booked for maintenance at the same time intervals and, as a result, the nett reserve of the system could fall below zero at these intervals. Such a solution-schedule would violate the constraints of Eq. (2.6) and would, therefore, be illegal.

However, the problem could be divided into several parts that are solved consequently one after another. It is possible to assign a few units their maintenance intervals, and

then gradually build up the schedule adding new units into it, similar to a schedule builder in an indirect representation as discussed in Chapter 2. In fact, if a human expert was to design a similar schedule, the general rule is to assign the units some priority value, based usually on the unit's capacity, and to then allocate the units' maintenance time one after another according to their priority.

3.2.2 *Introducing a multi-layered genetic algorithm*

The algorithm suggested in this thesis simulates the human expert approach. The units in the power system are divided into several groups according to their 'difficulty' for scheduling. First, the most difficult units are scheduled so that they provide a specified nett reserve throughout the entire period of maintenance. After that units from the next group are added, taking into consideration the sub-schedules that were obtained earlier.

This process could be seen as layering the GA, in that the algorithm divides a problem into layers and the final solution is obtained after solving a series of consecutive problems. Thus, the proposed algorithm is called a *multi-layered genetic algorithm* (MLGA). Each layer represents a separate GA that operates on a reduced search space, scheduling a limited number of units. Each GA layer starts running only after the previous GA layer has finished its job successfully, by finding a given number of sub-schedules that satisfy specified criteria. Each layer uses the sub-schedules obtained in the previous layer for building new, larger sub-schedules.

The MLGA therefore utilises the ideas represented by the search methods discussed earlier in this chapter. Firstly, the search in each layer is focused on some particular area of the problem domain, depending on the units that are being scheduled in that layer and the units that have been scheduled in the previous layers. Secondly, the sub-schedules found in the previous layer act as single genes in the next layer, or, in other words, these sub-schedules are preserved as good building blocks. As a result, a solution's structure evolves gradually during the execution of the MLGA.

The algorithm was reported in (Kelareva and Negnevitsky, 2001a; Kelareva and Negnevitsky, 2001b).

3.2.3 *Specifics of an MLGA search*

As was explained earlier, a GA with an indirect representation explores only a part of the problem domain that is restricted by a schedule builder, employed in the algorithm to transfer sequences of units into schedules. However, when an MLGA finds a number of solutions in each layer, by passing them as complete building blocks to the next layer the algorithm directs the search into particular areas. This way the MLGA also limits the search in subsequent layers and such limitation may result in converging to a local optimum and inability of the algorithm to find good solutions in subsequent layers.

The difference in the search restriction of the two algorithms is illustrated in the following two figures. Figure 3.1 presents the search space of a GA with an indirect representation within the problem domain. The region is defined by the schedule builder design and is exactly the same in every run. Although the individuals in the restricted region can be of good quality, the algorithm cannot examine individuals that are not included into the region because of the limited nature of a schedule builder. As a result the GA with an indirect representation may fail to find an optimal solution.

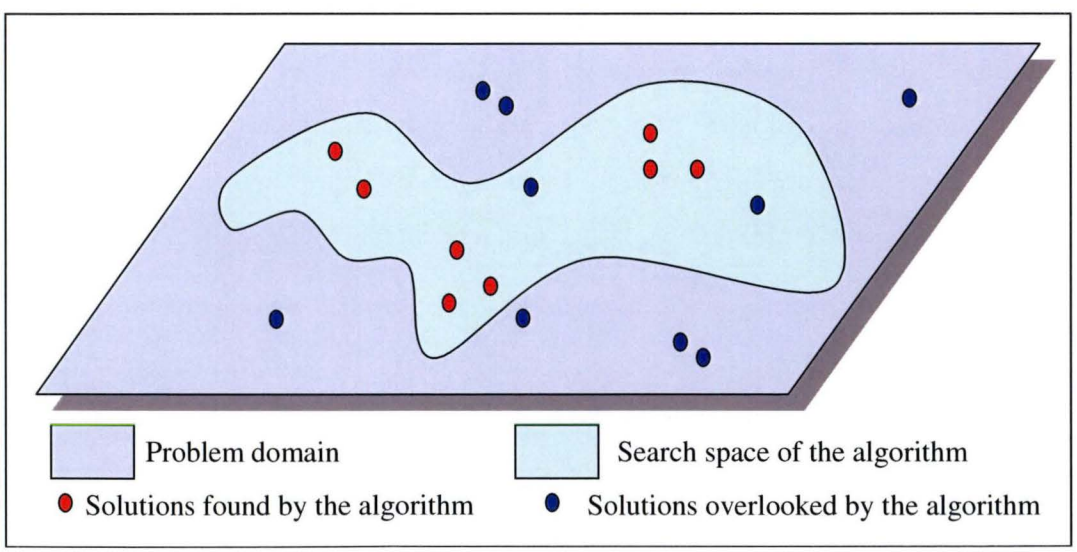

Figure 3.1 Search space of a GA with a schedule builder

In contrast, an MLGA explores the entire domain in the first layer, as shown in Figure 3.2. A randomised search allows the algorithm to find different solutions in the first layer every time it runs and the search in subsequent layers is performed in the regions

based around these solutions. This is an entirely different restriction of the search from the one imposed by a schedule builder, since the solutions and, consequently, the areas of search, vary from run to run. For example, even if a solution is not included into a search space of the second layer in one run, it still has a chance to be found by this layer in the next run, provided that the units from the first layer are scheduled in this solution according to specified criteria. This way the MLGA still has an opportunity to explore the entire domain of the problem even if in every single run its search space is restricted.

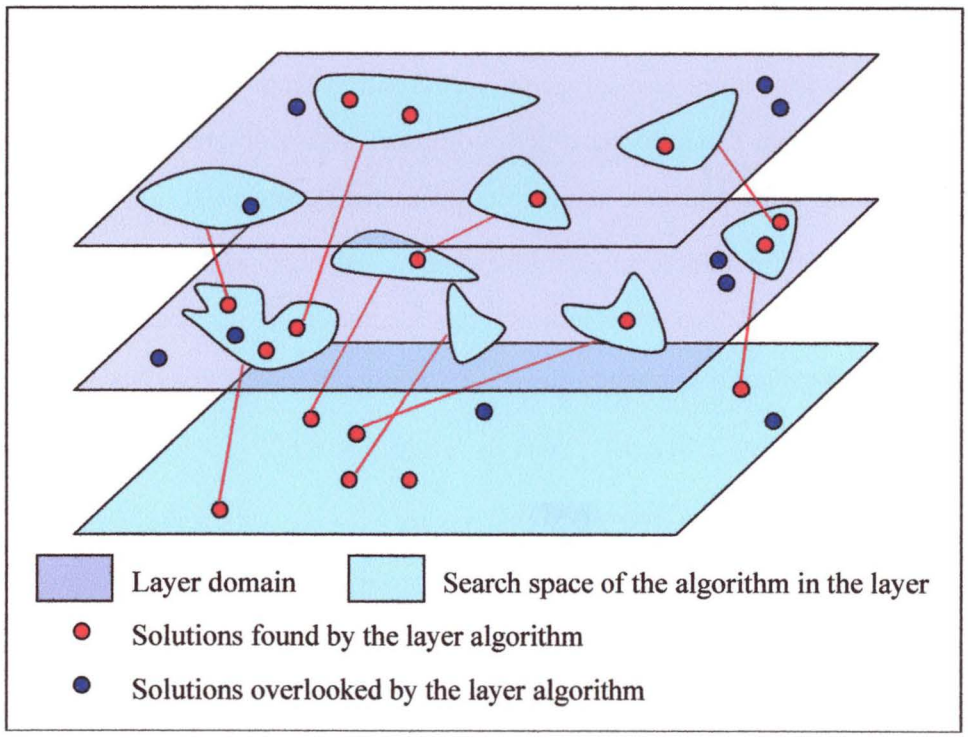

Figure 3.2 Search by an MLGA

Depending on the quality of the search strategy employed the MLGA will or will not be able to find good solutions in the restricted areas. There are several specific GA features incorporated in this type of algorithm, providing satisfactory convergence and at the same time promoting diversity in the population. These features, discussed in detail in the following sections, ensure that a GA in every layer gives adequate performance, providing the next GA layer with a gene pool of sub-schedules from which the next layer schedule can be built.

3.2.4 *MLGA parameters*

Poolwise representation with varying gene lengths

Suppose there are K layers in an MLGA, which means that N units have been divided into K groups, containing $N^{(k)}$ units in each group $k=1,\ldots,K$. Then each layer in the MLGA is a separate GA with a poolwise representation. That is, when a population is initialised, each gene is randomly assigned a value from a corresponding gene pool. The number of genes in a chromosome is different in each layer and depends on the number of units being scheduled in the particular layer.

All genes in the first layer and all genes representing single units in the consequent layers have a length of 1. These genes determine the start of the maintenance periods of the corresponding units and take their value from the gene pools $\mathcal{A}_j$, that are defined according to the maintenance requirements and units' capacities, as was discussed in Section 2.4.3. As in the case of indirect representation, a required retained reserve parameter, R^0, is given in the beginning of an experiment and a gene pool $\mathcal{A}_j$ for a unit j is calculated using Eq. (2.14). R^0 varies from 1220 to 1260.

In all MLGA layers except the first, that is, for $k=2,\ldots,K$, chromosomes contain genes of various lengths. One gene in every chromosome represents a sub-schedule built by the previous GA layer. This gene has a length equal to the number of units scheduled so far, and its value is taken from a gene pool containing all legal sub-schedules found by the previous MLGA layer, $\mathcal{S}^{(k-1)}$. The remaining genes in the chromosome are of length 1. They represent the units that are being scheduled in the current layer and take values from the corresponding gene pools $\mathcal{A}_j$. Thus an individual-schedule in layer k is described as following:

$$s^{(k)} = \left(a_1,\ldots,a_{N^{(k)}}, s^{(k-1)}\right), \tag{3.1}$$

where $a_j \in \mathcal{A}_j$ for all units $j=1,\ldots,N^{(k)}$ and $s^{(k-1)} \in \mathcal{S}^{(k-1)}$.

The genetic operators used by each GA layer are one or two-point crossover, depending on the number of genes in a chromosome in a particular layer; and a poolwise mutation which randomly changes a gene for another one from the same gene pool. Since an

elitist approach is implemented (see below), the crossover rate is taken to be 1. The mutation rate is traditionally small, and is set at 0.05 (Goldberg, 1989).

Gene pool criterion

To specify which individuals are included into the next layer's gene pool, a *pool threshold* should be defined. In the experiments conducted in this research, the pool threshold is assigned the same value as the required reserve parameter, R^0. Thus, a sub-schedule is included into the pool only if it can provide a nett reserve of at least R^0 at any time. Thus a criterion for the gene pool can be formulated as following:

Criterion 3.1 A sub-schedule $s^{(k)} = \left(a_1, \ldots, a_{N^{(k)}}, s^{(k-1)}\right)$ built in layer k is included into the $k+1$ layer gene pool $\mathcal{S}^{(k)}$ with the gene pool threshold R^0, if

$$f^{obj}\left(s^{(k)}\right) \geq R^0 \tag{3.2}$$

Individuals that satisfy the pool criterion/criteria are also called *good individuals* in this thesis.

Fitness evaluation and selection for reproduction

Since the individuals in the population represent only partial schedules, the objective and fitness functions also need to be modified. The aim of each GA layer is to schedule a number of units while retaining the nett reserve of the system of at least R^0, and the objective function in a layer is the same as in Eq. (2.5): $f^{obj} = \min\{R_i, i = 1, \ldots, 52\}$. The only difference is that the nett reserve, R_i, at a week i is calculated to be:

$$R_i = S - P_i - \sum_{j \in J_i^{(1,\ldots,k)}} C_j = G_i - \sum_{j \in J_i^{(1,\ldots,k)}} C_j, \tag{3.3}$$

where S is the system's capacity, P_i is the predicted load in week i, G_i is the gross reserve of the system in week i, C_j is the capacity of a unit j (see also Section 2.2.1) and $J_i^{(1,\ldots,k)}$ contains the units scheduled in week i from the current layer k, as well as from the previous layers, if there were any.

In Chapter 2, the fitness has been defined as the objective value provided by the individual, as in Eq. (2.9). When a proportional selection for reproduction is performed, the fitness is ranked according to Eq. (2.10), and for this purpose the fitness of an individual represents its rank in the population, based on the objective function value:

$$F_l = \begin{cases} rank(f_l^{obj}), & \text{if } f_l^{obj} \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

When the fitness function is defined as above, individuals with negative objective values, that is, not providing enough nett reserve at some interval and thus violating the problem's constraints, given by Eq. (2.6), are not considered for breeding.

As in the experiments described in Chapter 2, stochastic universal sampling, providing a proportional selection for reproduction, is used by the MLGA in each layer. This selection method promotes better individuals, while preserving the population diversity (Grefenstette, 1997a, 1997b).

To further diversify the GA population, stochastic universal sampling is replaced by random selection when there is a considerable amount of 'good' individuals present in the population (Eshelman and Schaffer, 1991). That is, all legal individuals are given an equal chance to be selected for reproduction by being assigned the same rank. This proved to be beneficial since proportional selection can slow the search for new 'good' solutions by choosing the same parents all the time.

Population size and replacement strategy with the growing elite

The aim of a particular GA layer is not just to reach a single optimal solution, but to find a number of them so that they can be used as separate genes for the next GA layer. The maximum number of solutions the algorithm attempts to find is equal to the population size, which was limited to 300 individuals for computational reasons.

Similarly to the previous chapter, an elitist approach with a varying generation gap was implemented in the MLGA, in order to obtain the desired number of sub-schedules faster (Eshelman and Schaffer, 1991). Initially the generation gap is equal to 0.9, but when the number of individuals suitable for inclusion into the pool becomes greater than 10% of the population, the generation gap is reduced. In other words, the elite part of the population is enlarged, to let all 'good' parents remain in the population until

their number equals the required size of the pool. At the end of the layer the whole population could be transferred into the next layer. Note that this approach saves computational time in a successful run since the number of individuals needing to be evaluated decreases over the generations.

To prevent premature convergence to a small number of good solutions, and to obtain a variety of sub-schedules in the gene pool for the next GA layer, new individuals are introduced into the population only if they are different from all the others already present (Syswerda, 1989; Whitley, 1989). If there are several identical individuals in the population, all but one of them are replaced by randomly built new individuals. This ensures regular introduction of new genes into the population, which improves its diversity (Eshelman and Schaffer, 1991; Sarma and De Jong, 1997).

3.3 MLGA implementation

3.3.1 Unit groupings for an MLGA

When the unit data in Table 2.1 is considered, it is clear that the units vary greatly in capacity, as well as the number of weeks required for maintenance. Unit capacities range from 20 MW to 460 MW, with the number of maintenance periods required ranging from 1 to 11. Obviously, units of a larger capacity with a greater maintenance period are more difficult to schedule than units of smaller capacity which need only one or two weeks for their maintenance.

For example, if the four units with a capacity of 460 MW are considered (No. 37, 38, 39 and 40), it is clear that no more than two of them should be scheduled for service simultaneously, if a required reserve parameter, R^0, is at least 1220. As shown in Table 2.2, the predicted load of the system is lowest at week 27, which means that the gross reserve of the system is at its highest, 2220 MW, at the same time. If three units with capacity 460 MW are switched off for maintenance, the nett reserve at week 27 will become $2220 - 460 \times 3 = 840$ MW, which is well below R^0.

There are also a few other units which are difficult to schedule; for example, a unit with a capacity of 230 MW and an 8 week maintenance period; the eleven other units with a

capacity of 210 MW requiring 4 week periods, and so on. Conversely, with some of the units that have a small capacity and/or a short maintenance period, it will be easier to find suitable combinations for several 'difficult' units first, thus reducing the search space, and then adding the other, 'easy', units to the schedule.

Table 3.1 replicates Table 2.1, except that the units that are considered difficult to schedule are displayed in bold, while the units that can be easily scheduled are shown in italic.

Table 3.1. Unit data

Unit #	Capacity (MW)	Maintenance requirements (weeks)	Unit #	Capacity (MW)	Maintenance requirements (weeks)	Unit #	Capacity (MW)	Maintenance requirements (weeks)
1	150	3	16	100	4	*31*	*30*	*3*
2	150	3	17	100	4	*32*	*30*	*3*
3	150	3	**18**	**200**	**4**	*33*	*20*	*2*
4	**210**	**4**	**19**	**200**	**4**	*34*	*80*	*1*
5	**210**	**4**	**20**	**210**	**4**	*35*	*80*	*1*
6	**210**	**4**	21	100	5	36	130	3
7	**210**	**4**	22	50	4	**37**	**460**	**8**
8	**210**	**4**	23	60	4	**38**	**460**	**11**
9	**210**	**4**	24	60	4	**39**	**460**	**6**
10	**230**	**8**	25	60	4	**40**	**460**	**4**
11	160	2	26	30	7	41	120	1
12	**210**	**4**	27	30	7	42	120	1
13	**210**	**4**	28	60	7	43	170	1
14	**210**	**4**	29	60	7			
15	**210**	**4**	*30*	*30*	*3*			

Note: 'Difficult' units are shown in bold, 'easy' units are shown in italic.

A number of experiments were conducted in order to determine a suitable grouping of the units. Special attention was paid to identifying the units that should be included in the first layer. Table 3.2 summarises the results of some of these experiments. The entries in the table are each the average of 20 runs.

The experiments are divided into several categories according to the number of units being scheduled, and the pool threshold value, R^0. Each experiment was run with 300 individuals in the population. The algorithm was stopped after 300 generations unless the next layer pool was filled earlier, that is, the population consisted only of individuals with fitness no less than R^0.

Various groups of units were tried for scheduling; Table 3.2 presents the experiments with the following groups:

1. 15 units with the capacity of 460 and 210 MW; No. 4 to 9, 12 to 15, 20 and 37 to 40.
2. Unit No. 10 with the capacity of 230 MW is added to Group 1.
3. Group 2 with two more units added, No. 18 and 19. Their capacity is 200 MW each.
4. The 18 units in Group 3 are scheduled in two layers: Group 1, in the first layer; while units No. 10, 18 and 19 are added in the second layer.

The key parameters shown in Table 3.2 are:

- Number of generations in the run.
- Number of generations required to find the first individual with fitness no less than the pool threshold, R^0.
- Best and average fitness values in the first and the last generations.
- Number of individuals/schedules with fitness at least equal to the pool threshold value, R^0, in column j and greater than that value, in the last generation.

Groups 1 and 2

As can be seen in the table, 15 or 16 units are scheduled easily during all twenty runs. It takes an average of 4.6 to 9.2 generations to find the first suitable schedule in the first case, and 18 to 38.7 in the second case, taking usually a little longer to find the schedules satisfying the higher pool threshold.

The algorithm is able to find not only schedules with the desired reserve, but also with a better one. For example, the best schedule found for the 15 units always provides the maximum reserve of 1260 MW; the same is true for all experiments with 16 units and threshold equalling 1250, and for almost all with a threshold less than 1250.

Group 3

The scheduling of the 18 units in the third group of experiments is not as successful, especially with the gene pool threshold, R^0, equal to 1250 and 1260. The algorithm failed to find a single individual with fitness equal to R^0 or better in 3 out of 20 runs with R^0=1250; and in 7 out of 20 runs with R^0=1260. Even when the algorithm is successful, it takes 47.4 to 86.3 generations to find the first suitable schedule, which is about ten times slower than the corresponding results with 15 units, and more than twice as slow than the results with 16 units. The number of schedules which provide a reserve better than the threshold value is also lower.

Table 3.2 Results of the partial GA experiments

Scheduling group / Number of units	Gene pool threshold, R^0	Number of successful runs out of 20	Number of generations in the run	Generation of the first ind. with reserve => R^0	First generation		Last generation			
					Best fitness	Average fitness	Best fitness	Average fitness	Number of inds. with fitness >= R^0	Number of inds. with fitness > R^0
1 / 15	1220	20 (100%)	57.10	4.60	1152.50	649.70	1260.00	1225.56	300	226.00
	1230	20 (100%)	59.35	5.35	1147.50	651.30	1260.00	1233.81	300	217.10
	1240	20 (100%)	56.40	7.05	1155.00	651.29	1260.00	1239.86	300	152.15
	1250	20 (100%)	60.40	8.75	1147.50	649.70	1260.00	1249.45	300	242.90
	1260	20 (100%)	65.45	9.20	1143.00	644.16	1260.00	1250.73	300	N/A
2 / 16	1220	20 (100%)	84.35	17.95	1088.00	572.99	1256.50	1218.85	300	149.45
	1230	20 (100%)	90.90	21.70	1083.50	566.82	1256.50	1230.57	300	193.35
	1240	20 (100%)	91.70	20.25	1092.00	569.93	1259.00	1234.76	300	91.45
	1250	20 (100%)	103.15	26.40	1088.00	572.41	1260.00	1242.50	300	138.10
	1260	20 (100%)	123.50	38.70	1080.00	569.39	1260.00	1248.92	300	N/A
3 / 18	1220	20 (100%)	126.80	47.40	1031.50	497.60	1247.00	1215.00	300	109.45
	1230	20(100%)	159.30	59.70	1046.00	511.40	1256.50	1225.87	300	126.15
	1240	20 (100%)	183.95	82.35	1060.00	506.78	1252.50	1226.00	294.50	63.50
	1250	17 (85%)	216.30	76.85	1059.50	503.96	1257.00	1171.68	240.45	87.85
	1260	13 (65%)	270.20	86.30	1052.00	494.12	1254.00	1075.13	153.90	N/A
4 / 15+3	1220	20 (100%)	163.20	2.90	1191.00	971.12	1246.50	1215.50	295.85	113.35
	1230	20 (100%)	190.70	4.00	1198.50	971.12	1250.50	1204.78	266.85	106.95
	1240	20 (100%)	224.30	11.75	1203.00	979.88	1252.50	1162.59	198.75	37.20
	1250	20 (100%)	253.40	15.90	1194.00	982.66	1259.00	1141.35	159.65	49.40
	1260	20 (100%)	260.05	20.30	1212.50	982.39	1260.00	1115.52	119.50	N/A

Group 4

When the same 18 units are scheduled in two layers, the success rate is 100% again. Table 3.2 shows the final results of the second layer, except for the number of generations in a run (column *d*), which is the sum of the generations in both layers. It takes only a few generations, 2.9 to 20.3, to find the first satisfactory schedule in the second layer when the units No. 10, 18 and 19 are added to the pre-scheduled 15 units, which demonstrates that dividing units into groups for scheduling has some advantage.

The experiments presented in Table 3.2 show that 15 or 16 units from Groups 1 and 2 can be scheduled successfully in the first layer. What remains is to allocate the rest of the units into subsequent layers. A number of different groupings were trialed and the most promising initial results, reported in (Kelareva and Negnevitsky, 2001a), came from a 12-layer MLGA with the units divided into layers as shown in Table 3.3. The first two layers represent the 18 units scheduled in the Group 4 experiments described above. After that, two or three new units are added to the schedule on each subsequent layer. The results of a MLGA with this grouping are discussed in detail in Sections 3.4 and 3.5. The number of genes in each chromosome and the corresponding gene length are also given in Table 3.3.

Table 3.3. 12-layer MLGA

Layer	Units being scheduled	Number of genes	Gene length
1	4 5 6 7 8 9 12 13 14 15 20 37 38 39 40	15	1
2	10 18 19	4	1,1,1,15
3	1 2 3	4	1,1,1,18
4	11 21 36	4	1,1,1,21
5	16 17	3	1,1,24
6	28 29	3	1,1,26
7	24 25	3	1,1,28
8	26 27	3	1,1,30
9	22 23	3	1,1,32
10	41 42 43	4	1,1,1,34
11	30 31 32	4	1,1,1,37
12	33 34 35	4	1,1,1,40

3.3.2 *Interchangeable chromosomes*

In Chapter 2, it was mentioned that in the case study problem, some individuals/chromosomes, which are different on the gene level, can represent similar schedules. This is because a number of units have exactly the same capacity, and the number of required maintenance intervals can be rearranged without making any difference to the nett reserve of the schedule.

For example, consider two chromosomes representing sub-schedules for the six units No. 4, 5, 6, 7, 37 and 38:

Chromosome 1: {12 15 16 30 20 25}
Chromosome 2: {30 16 15 12 20 25}

Although genes 1 to 4 are different in the two chromosomes, they describe the beginning of a maintenance outage for units 4 to 7, which, according to Table 3.1, have the same capacity, 210 MW, and require the same amount of maintenance, four weeks. That means that the chromosomes represent interchangeable sub-schedules, as the four genes can be rearranged in any order without making any difference to the system's nett reserve at any time interval. Therefore, every chromosome containing several equivalent units represents a group of interchangeable schedules.

However, if a crossover operator with the crossover point between the first and the fourth genes is applied to the two chromosomes from the previous example, the resulting chromosomes are completely different. For example, if the crossover point is between the second and the third genes, the result of the crossover is as following:

Parent 1: {12 15 | 16 30 20 25}
Parent 2: {30 16 | 15 12 20 25}

Offspring 1: {30 16 | 16 30 20 25}
Offspring 2: {12 15 | 15 12 20 25}

According to the principle of minimal redundancy, each member of the problem domain should be represented by as few different chromosomes as possible, ideally only one (Radcliffe, 1991). It can be argued that interchangeable individuals should be discarded from the population since they represent the same schedule. Yet, since a crossover

operator applied to two interchangeable individuals may still produce non-interchangeable offspring, all interchangeable individuals have been allowed to stay in the population.

The issue of interchangeable individuals has a different meaning when a gene pool for the next layer is considered. Before passing the gene pool to the next layer, all the sub-schedules are checked and interchangeable sub-schedules are all eliminated except one. This minimises the representation redundancy in the next layer since all sub-schedules found so far are represented by only one gene in the next layer's gene pool.

3.3.3 Unit convergence

By choosing a number of sub-schedules and gradually adding new units to them, an MLGA focuses the search on a few promising regions within a large problem domain and finds solutions faster than a traditional GA. On the other hand, since the search is restricted to the sub-schedules found in the previous layers, some units can converge, that is, they can be scheduled exactly in the same way in all individuals. (Note that because the duplicates are eliminated, the population will never wholly converge.) This unit convergence is similar to the gene convergence, a term used to describe a case when a gene has the same value in a specified high percentage of the population (Michalewicz, 1999). However, since the converged units may represent only a part of the large gene in a chromosome it would be more appropriate to use a term 'unit convergence' instead of 'gene convergence', when an MLGA is considered.

If some units have converged and all genes in a gene pool contain the same sub-schedule, all individuals in the next layers will contain this sub-schedule. As a result, if potentially good sub-schedules were not developed into larger successful sub-schedules and were consequently lost in some layer, the entire run may fail. This thesis does not intend to discuss various existing GA techniques to reduce unit convergence, such as increasing the mutation rate, etc., investigated by various researchers (Back, 1993; Tate and Smith, 1993). Instead, this potential disadvantage of unit convergence can be used to the benefit of the further search. For example, it is possible to modify the sets of

allowed values for the remaining units once more, thus reducing the search region. This procedure is similar to the one discussed in Section 2.4.3, except that the converged part of the gene pool is treated as an additional load on the system.

Suppose that all genes in a gene pool formed in the previous layer contain a sub-schedule $s=(a_1,a_2,\ldots,a_m)$, allotting maintenance intervals to m units, which means that some of the system's resources are already "booked" by these units. Then the additional load on the system in an interval i will be

$$L_i=\sum_{j=1}^{m} C_j X_{ij}, \tag{3.5}$$

where C_j is the capacity of unit j, while $X_{ij}=1$ if $j\in J_i$, that is, if maintenance is performed on unit j in interval i, and 0 otherwise. Modifying Eq. (2.13), we obtain the new gross reserve of the system

$$G_i''=S-P_i-R^0-L_i \tag{3.6}$$

and find the new sets of allowed numbers A_j for the units that should be scheduled in the next layer, substituting G_i'' instead of G_i' into Eq. (2.14). This procedure only needs to be done once at the beginning of every layer and, therefore, does not increase the computational time.

3.3.4 Termination criteria

The foremost termination criterion is filling the gene pool for the next layer, that is, reaching a specified number of suitable solutions. Therefore, the algorithm is stopped when all individuals in the population have fitness equal to or greater than the gene pool threshold, R^0. If after a given number of generations the gene pool is not filled, the algorithm can be terminated when there is no significant improvement in its performance (Michalewicz, 1999). For this reason, the GA performance is assessed with respect to the speed of the pool being filled. If there is no change in the number of 'good' individuals to be included in the pool during a specified number of generations,

the layer is considered completed and, if there are no sub-schedules found, the run is declared unsuccessful.

This termination criterion, concerning the algorithm's potential for improvement, needs some additional consideration. Obviously, the faster the pool is filled, the better the GA in this layer works. However, sometimes the pool is filled very slowly, either because of the difficulty of this particular layer; or if all suitable sub-schedules have been found, but their number is still less than the desired size of the pool. In the first case, the algorithm needs more time to find as many sub-schedules as possible; while in the second case, if the pool is mostly filled, the algorithm could be stopped without any loss of efficiency.

Therefore it was suggested that after $\mathcal{T}_1$ generations the size of the pool is checked and, if in the last $\mathcal{T}_2$ generations it's growth was less than 10%, the algorithm is stopped. If there is still considerable growth in the size of the pool, it is checked every $\mathcal{T}_3$ generations till the maximum number of generations is reached. This allows the algorithm to be stopped in the situation when the pool is almost filled but there are scarcely any more suitable sub-schedules left. On the other hand, if a layer is a particularly difficult one, taking many generations to find the first suitable individual and slowly filling the pool, the algorithm is allowed to run while some progress is still being made.

The termination parameters $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ were extensively tried. The first parameter is the most important, if it is small, it can reduce the running time in some layers but affect the overall performance by decreasing the GA chances of succeeding. After numerous tests the parameters were assigned values of 200, 20 and 20 respectively as providing the most effective GA performance. The maximum number of generations in one layer is restricted to 400.

3.3.5 *Pseudo code for an MLGA*

Figure 3.3 illustrates the MLGA method discussed in the previous sections. After defining the pool criteria and dividing the problem into layers, an ordinary GA is employed in each layer. All individuals satisfying pool criteria are allowed to stay in the

population. For this reason the size of the pool is calculated and the generation gap is adjusted every generation if new good solutions are found.

```
begin MLGA
  define Pool_criteria        % define Pool criteria
  define J^(k), k=1,...,K     % define K layers
  S^(0) = empty   % no sub-schedules in the gene pool for the first layer
  for every k
  t=0
  S^(k) = empty               % gene pool for the next layer
  define Pop_size             % define population size (300)
  define Ggap                 % define generation gap value (0.9)
  P(t) =initialise( J^(k), S^(k-1) )     % initialise population with genes from J^(k)
                                         % and S^(k-1) if not empty
  F(t) =evaluate ( P(t) )
  Pool_size=size(P(t)|Pool_criteria)     % number of individuals suitable for S^(k)
  Ggap=min(Ggap,(Pop_size-Pool_size)/Pop_size)
  until (termination_criteria) do
      t=t+1
      P'(t-1) =select( P(t-1),Ggap)  % select Ggap*Pop_size individuals
                                     % for reproduction from P(t-1)
      P'(t) =recombine_and_mutate( P'(t-1) )
      P(t) =replace( P(t-1), P'(t) ,Ggap)  % replace Ggap*Pop_size individuals
                                           % from P(t-1) by P'(t)
      F(t) =evaluate( P(t) )
      Pool_size=size( P(t) |Pool_criteria) % number of individuals suitable for S^(k)
      Ggap=min(Ggap,(Pop_size-Pool_size)/Pop_size)
  end  % until loop
  S^(k) = select( P(t),Pool_criteria)   % part of P(t) satisfying Pool_criteria is
                                        % transferred into the next layer pool, S^(k)
  if S^(k)=empty, end MLGA  % no solutions found
  end  % for k loop
end MLGA
```

Figure 3.3 Pseudo code for an MLGA

3.3.6 *Increasing the population size for better performance*

The algorithm stops when all 300 individuals in the population have a fitness no less than R^0 and, therefore, the whole population can be included in the next layer gene pool. In practice however, this approach may slow down the algorithm because at the end of a successful run, when the gene pool is almost full, new individuals are harder to find.

For example, suppose there are 297 individuals that satisfy the reserve threshold criteria and are therefore placed in the pool. These 297 individuals would now stay in the population forever, and the only part of the population that will change every generation is the three individuals that don't satisfy the threshold criteria. To fill the outstanding three places in the pool, three parents should be chosen for reproduction from the current population of 300. Strictly speaking, an even number of individuals should be selected for crossover, but in order to generalise the procedure one randomly chosen parent simply reproduces itself and, possibly, gets changed in the next step by mutation. Most likely, the selected parents would be the ones from the pool that have been in the population for several generations and have had many chances to reproduce already. That might result in successful offspring being identical to the individuals that are already in the pool. In that case they are automatically discarded and replaced by randomly built new individuals with, possibly, some new genes that are not in the population. But there is only a 3/300=0.01 chance that one of these three new individuals will be selected for reproduction.

As demonstrated by the experiments described in Table 3.4 below, the process of finding the last few individuals for the pool might take a considerable amount of time. To avoid this, the population size could be increased up to 320. That would mean, of course, an increase in the processing time of every population, mostly due to evaluation of additional individuals. However, the population could equal 300 most of the time, and only when, say, 70 % of the pool is filled, would the population be increased by adding a number of randomly built individuals. That number of additional chromosomes could be the parameters of a particular GA. Another possible approach would be to stop the algorithm when most of the pool is filled, then the number of genes in the next layer gene pool will be less than 300.

Consider a situation such as that described in the previous paragraph, when there are 297 solutions in the pool already and the population size is increased to 320. Then the chance of a non-pool individual being selected for breeding increases to 23/320=0.072, which is seven times more than in the population of 300. Besides, 23 new individuals will be created instead of only three as in the previous case, which makes choosing the last three individuals suitable for the pool easier. In addition, if more than 300 individuals that satisfy the reserve threshold criteria are found, they all are included into the pool.

Table 3.4 summarises the results of an experiment concerning the population size increase. The experiment examines the performance of the first GA layer scheduling fifteen genes. The population size was taken to be 300, 310 and 320 individuals. The required pool size for the next level was 300 in all experiments. There were 20 runs of each experiment and the entries in the table are the averages of these runs.

Table 3.4 Effect of the increase in the population size on MLGA performance

Population size	Number of generations to fill the gene pool			Number of generations in the run	Generation of the first 'good' individual
	from 250 to 300	from 275 to 300	from 290 to 300		
300	26.45	21.60	15.65	66.80	5.20
310	13.50	9.35	5.45	55.15	4.80
320	9.55	6.30	3.35	49.85	4.75

As seen from the results in Table 3.4, a GA with an enlarged population finds the last few individuals suitable for the next layer pool faster than a GA with a population of only 300 individuals. If the population size is increased by 10, it reduces by half the number of generations to find the final 50 solutions for the pool, while by adding 20 individuals the number of generations is reduced almost three times. The reduction in processing time is even more noticeable while finding the last 10 genes for the gene pool, with the number of generations being reduced almost three and five times respectively. In fact, GAs with a population size of 300 spend an average of 6 generations to find the last, 300^{th} solution, while GAs with larger populations find the

last five or so solutions in only one generation. By increasing the population by 3 or 7 %, the length of a run is reduced by more than 17 and 25 % correspondingly.

It was found that if the population size is set at 310 or 320 from the beginning of the run, there is only a small difference in the number of generations needed to find the first 'good' individual, as can be seen from Table 3.4. Therefore, if evaluation of additional individuals is considered time consuming, the population could be increased only at the end of the run.

Figure 3.4 shows an example of GA performance with 300 individuals in the population. It took 29 generations to find genes 290 to 299 and another 19 to find the last gene for the pool.

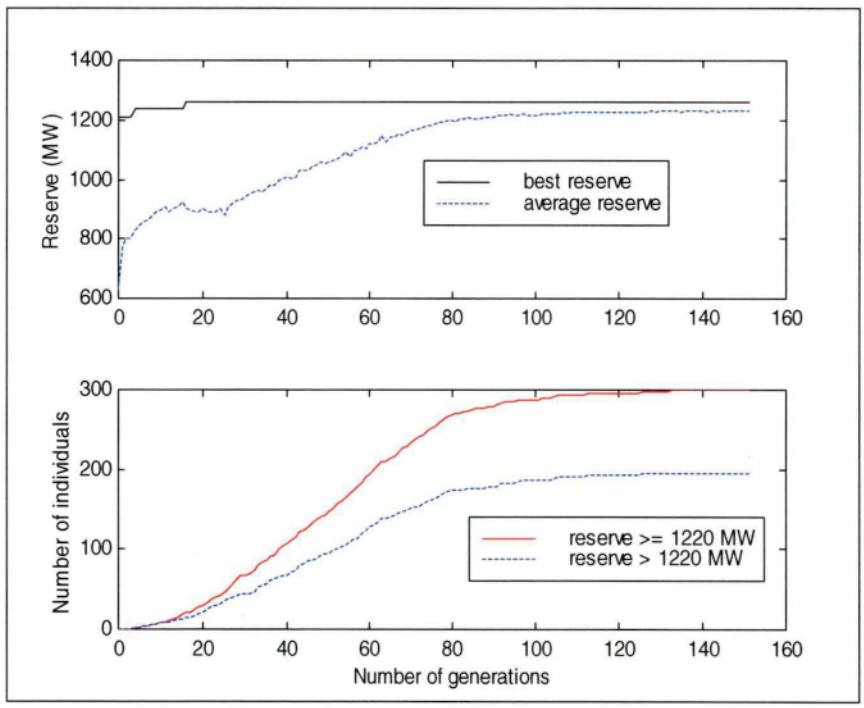

Figure 3.4 Example of GA performance with population size 300, pool size 300.

The increase in population size in the later part of each run proved to be beneficial. Therefore, further analysis was based on a population size of 300 that was increased at the end of the run by 20 individuals in order to speed up the search for the last few individuals.

3.4 Preliminary results

3.4.1 Performance graphs

Appendix 2 contains Figures A2.1 to A2.12, representing performance graphs of an MLGA with twelve layers, each figure describing a corresponding layer. The figures demonstrate MLGA performance with the gene pool threshold, R^0, equal to 1230. Each figure consists of two graphs, the first graph describing the best and average objective value in the population over the generations, while the second graph shows the number of sub-schedules in the pool, that is, the number of individuals in the population with an objective value of at least 1230. Another parameter presented in the second graph is the number of individuals with an objective value greater than the pool threshold.

The performance graphs show a characteristic picture of the progress of an MLGA. Several features of the algorithm can be discerned from the graphs.

- In five of the layers an individual providing the objective value of 1230 MW was present in the initial population (layers No. 5, 7, 8, 10 and 12), while in the other layers it took up to twelve generations to find the first 'good' individual.
- Most of the layers took less than 100 generations to find 300 or more solutions for the next layer pool.
- The 'good' solutions show almost exponential growth, slow at first and growing faster, until their number is comparable with the size of the population itself. At this stage the growth slows as new additions to the pool are found less frequently (see Figures A2.1 and A2.4, for example). The other possible reason for the number of sub-schedules growing slowly at the end of the layers No. 6 and 9 could be a shortage of suitable individuals in the population, since these two layers were the only ones in the run where the pool was not filled completely.
- The effect of the termination criteria discussed in Section 3.3.4, can be demonstrated on the graphs presented in Figures A2.6 and A2.9. In the 9^{th} layer the pool was almost full by the end of 200 generations, and the execution of the layer stopped because there was no significant growth in the last 20 generations. At the same time, the pool of the 6^{th} layer had only 80 sub-schedules in it when the layer was terminated. As was said before, the minimal number of generations to be run, $\mathcal{T}_1$, is

an important parameter, and if the 6th GA layer was allowed to run longer, it might have found more sub-schedules for the pool. On the other hand, it might take an unnecessarily long time, as in Layer No.2, when the last 20 sub-schedules for the pool took 40 generations to find. Layer No.6 and, in particular, Layer No.9 proved to be the most difficult ones to complete in all experiments.

- Another typical feature of an MLGA is the presence of individuals with an objective value greater than the pool threshold in the first layers. This feature can be seen in Figures A2.1 to A2.4, where a significant part of the pool consisted of sub-schedules providing the nett reserve between 1240 and 1260 MW. Even later, in layers No. 5 to 7 there were a number of sub-schedules with the reserve of 1240 MW.

3.4.2 Evaluation of the 12-layer MLGA

Appendix 3 contains Tables A3.1 to A3.5, with detailed data summarising the performance of the 12-layer GA described above. Experiments were conducted with the pool threshold value ranging from 1220 to 1260 MW, and each experiment was performed forty times. Every layer's performance is described separately and every entry in the tables is an average of up to forty values, depending on the number of runs in which a particular layer took part. For example, if a layer successfully finished only ten times, all the consecutive layers are run no more than ten times and the corresponding entry is the average of no more than ten values.

Parameters presented in the table are as follows:

- Number of successful runs out of forty.
- Number of generations in each layer. This parameter is given separately for the successful runs and overall. That is, if the algorithm is unable to find successful solutions in a run, the layers from this run are not included in the number of generations in successful runs, even if a single layer was successful.
- Best and average objective values in the beginning of the run, taken over all runs.
- Generation in which the first solution, suitable for the next layer pool, or, in the last layer, a final schedule, was found. The tables contain the average value over successful layers, that is, if no solutions were found in a layer and, therefore, the

generation of the first solution is zero, it is ignored and not included in the average value.

- Best and average objective values at the end of the run. The best value is represented separately for the successful runs and overall, as for the number of generations in a run.
- Number of non-interchangeable sub-schedules in the next layer pool, if any, at the end of the layer run. In the case of the last layer, this parameter is the number of final non-interchangeable schedules found by the MLGA.
- Number of converged units in the population at the end of the run. This parameter gives more information about the population diversity than the traditionally measured number of converged genes, because in an MLGA one gene may contain several units.

As can be expected, the algorithm operates better when the threshold value, R^0, is lower. A 12-layer GA finds schedules providing at least 1220 MW of the nett reserve in 34 runs out of 40. When the threshold is 1230 MW and 1240 MW suitable schedules are found in 27 and 2 runs respectively. There were no schedules found that could provide a nett reserve greater than 1240 MW.

There are some similarities in GA performance in the different layers, no matter what the gene pool threshold is.

- For example, the 9^{th} layer seems to consistently be the most difficult layer, taking a large number of generations to find the required amount of solutions for the next layer pool. This factor proved to be critical for the entire algorithm, since if any solutions were found in that layer at all, then the last three layers would be successful as well. In fact, when the threshold value was 1240 MW, the two successful 9^{th} layer runs produced only 37 and 6 individuals suitable for the 10^{th} layer gene pool. The minimal size of the gene pool in the experiments with thresholds of 1220 and 1230 MW were 10 and 6 respectively. In spite of such a small number of sub-schedules found in layer 9, the 10^{th} GA layer was still able to find a significant amount of new solutions.
- The 10^{th} layer and, in particular the 12^{th}, prove to be the fastest, needing only a few generations to find the required amount of schedules that provide the nett reserve no less than the given threshold. Other fast layers are the 3^{rd} and 11^{th}.

- If the best value at the end of successful runs is examined, it is clear that the first layers produce solutions with an objective value greater than the required threshold. For example, solutions with an objective value of 1260 MW were found in the first two layers in all experiments. The layers when no solutions were found with the objective value greater than the threshold, were layer 10 for the threshold of 1220 MW, and layers 8 and 6 for the thresholds of 1230 and 1240 MW respectively.

  It should be noted that the existence of individuals with an objective value greater than the threshold does not guarantee the overall success of the algorithm. For example, in Table 3.8, the best objective value at the end of the successful runs is slightly less than the same parameter taken for all runs in layers 2 and 4. That means that in some of the unsuccessful runs these layers produced more sub-schedules with better objective values.

- Another noticeable tendency over the layers is the growing number of converged units. That is, all schedules found in the last layer have the same units scheduled for maintenance during the same weeks. By reducing the sets of allowed values, $\mathcal{A}_j$, as discussed in Section 3.3.3, this tendency could be used to speed up the search.

To summarise, the results of a 12 layer MLGA are better than those of the traditional GA examined in Section 2.4.3. Depending on R^0, the MLGA is able to find schedules providing a nett reserve of 1220 MW and 1230 MW in 34 and 27 runs out of 40. However, the MLGA only occasionally finds schedules with a nett reserve of 1240 MW, i.e. 2 runs out of 40. The indirect representation with the 'first available' schedule builder can still perform better than a MLGA (see Section 2.4.2), but it should be noted that a MLGA, when successful, finds about 300 different schedules while a schedule builder produces a large amount of identical schedules, as discussed in Section 2.4.4.

3.5 Modified gene pool criterion

3.5.1 Additional evaluation of gene pool candidates

The initial step in implementing an MLGA is choosing the units to be scheduled in the first layer, so that a traditional GA can find satisfactory sub-schedules. However, in subsequent layers, the algorithm sometimes fails to find solutions, even if there was a

substantial amount of genes in the gene pool. In this situation the entire run would be unsuccessful.

In order to improve the overall performance of an MLGA, the gene pool Criterion 3.1, given by Eq. (3.2), could be modified so that sub-schedules can be assessed as to whether they can provide enough reserve for the next layer units to be scheduled, before the scheduling actually takes place. This procedure was suggested in (Kelareva and Negnevitsky, 2001b) and is similar to finding the allowed values for genes discussed in Sections 2.4.3 and 3.3.3. The algorithm can be modified so that the genes are included in the pool only if they can provide enough of a reserve for the next layer's units to be scheduled.

In this procedure, every candidate for the gene pool $\mathcal{S}^{(k)}$, that is, every sub-schedule from layer k that provides the necessary minimal nett reserve R^0, is examined and treated as an additional load on the system L_t, as in Eq. (3.5). A new modified gross reserve, G_t'', is calculated, using Eq. (3.6), and the new sets of allowed numbers $\mathcal{A}_j'$ are found by Eq.(2.14) for the units $j \in J^{(k+1)}$, scheduled in the next layer. If neither of the sets $\mathcal{A}_j'$ are empty, it means that there exists at least one interval available for scheduling the next layer's units, and the sub-schedule is marked as suitable for the next layer gene pool. Thus, the modified gene pool criterion can be formulated as following:

Criterion 3.2 A sub-schedule from a layer k, $s^{(k)} = \left\{a_1, \ldots, a_{N^{(k)}}, s^{(k-1)}\right\}$ is included into the $k+1$ layer gene pool, $\mathcal{S}^{(k)}$ with the pool threshold R^0, if the following conditions hold:

$$\begin{aligned} &1)\ f^{obj}\left(s^{(k)}\right) \geq R^0 \text{ and} \\ &2)\ \mathcal{A}_j' = \{1, \ldots, 52 - M_j + 1\} \cap \left\{i \in \{1, \ldots, 52\} \mid G_t'' \geq C_j\right\} \neq \varnothing \text{ for all } j \in J^{(k+1)} \end{aligned} \tag{3.7}$$

If, on the other hand, any of the sets $\mathcal{A}_j$ are empty, it would mean that some of the units in the next layer cannot be scheduled as there are not enough maintenance intervals available for them. In this case the sub-schedule should not be included in the next layer gene pool, even if its fitness is equal to or above the pool threshold, R^0.

3.5.2 *Modified fitness function*

Individuals that satisfy Criterion 3.2 represent sub-schedules of better quality than the rest of the population, regardless of the objective value. This should be taken into account when a proportional selection operator chooses individuals for reproduction according to the individual's rank in the population, as in Eq. (3.4). For example, a bonus could be added to the objective value when calculating the rank of an individual satisfying Criterion 3.2 and, therefore, included into the pool. Thus, the new ranking procedure performed before the proportional selection takes place would be as following:

$$F_l' = \begin{cases} rank(f_l^{obj} + b), \text{ if } f_l^{obj} \geq 0 \text{ and } \mathcal{A}_j' \neq \varnothing \text{ for all } j \in J^{(next)} \\ rank(f_l^{obj}), \text{ if } f_l^{obj} \geq 0 \text{ and } \mathcal{A}_j' = \varnothing \text{ for some } j \in J^{(next)} \\ 0, \text{ if } f_l^{obj} < 0 \end{cases}, \qquad (3.8)$$

where b is a constant bonus value and $J^{(next)}$ contains units schedules in the next layer.

For example, in the experiments conducted in this research, b=50 would ensure that individuals with an objective value of 1220 and satisfying Criterion 3.2, would have a higher rank than individuals with an objective value of 1260 but not satisfying the criterion. Therefore, when a proportional selection is used, the individuals satisfying Criterion 3.2 would have a better chance to reproduce.

3.5.3 *Effect of the modified pool criterion on MLGA performance*

It could be argued that the use of the gene pool Criterion 3.2 expressed in Eq. (3.7) would mean an increase in computational time due to the additional check for every potential candidate for the gene pool. Yet the use of this criterion would simplify the search in the next layer by providing genes of a better quality. Alternatively, if no sub-schedules suitable for the pool were found, the run would be declared unsuccessful

without running the extra layer, even if some sub-schedules with an objective value of R^0 were found.

The effect of Criterion 3.2 was discussed in (Kelareva and Negnevitsky, 2001b) and is shown in Figures 3.5 and 3.6. The first figure shows a typical performance graph of the fourth GA layer with the pool threshold $R^0 = 1240\,\text{MW}$. The upper graph in Figure 3.5 represents the dynamics of the best and average fitness value in the population, that is, the minimal nett reserve provided by the sub-schedules. The lower graph shows the number of individuals in the next layer pool and, as a separate parameter, the number of individuals with fitness equal to the pool threshold, but not included in the pool since they can't provide the next units with a chance to be scheduled.

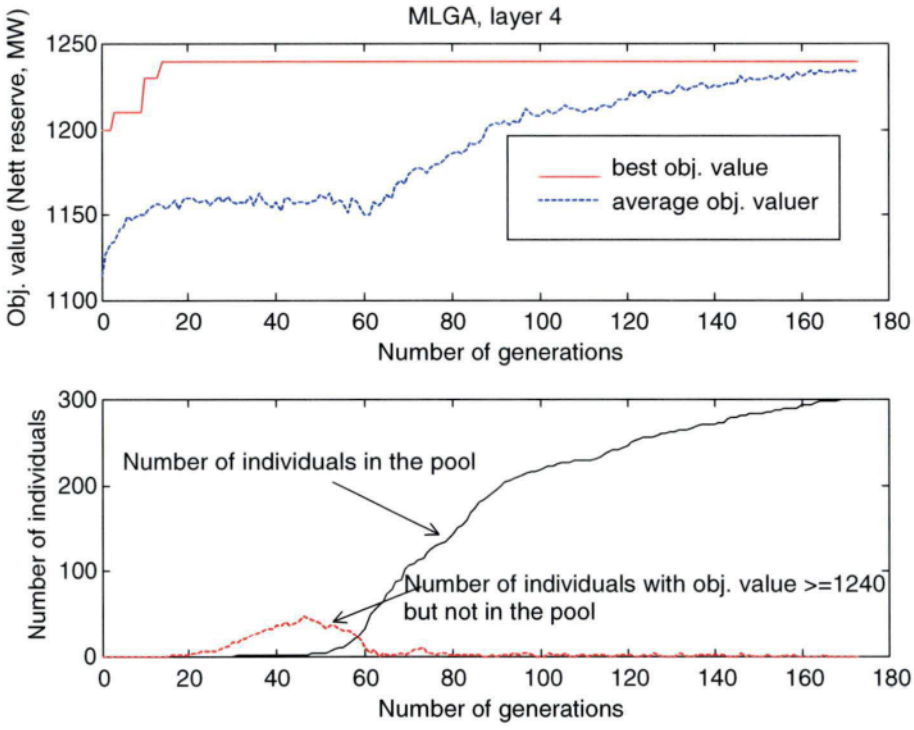

Figure 3.5 Performance of a GA with gene pool Criterion 3.2

Figure 3.6 shows the second graph from the previous figure in more detail. It presents the algorithm's performance during the first 70 generations. At the beginning of the run a significant number of otherwise good individuals did not satisfy Criterion 3.2.

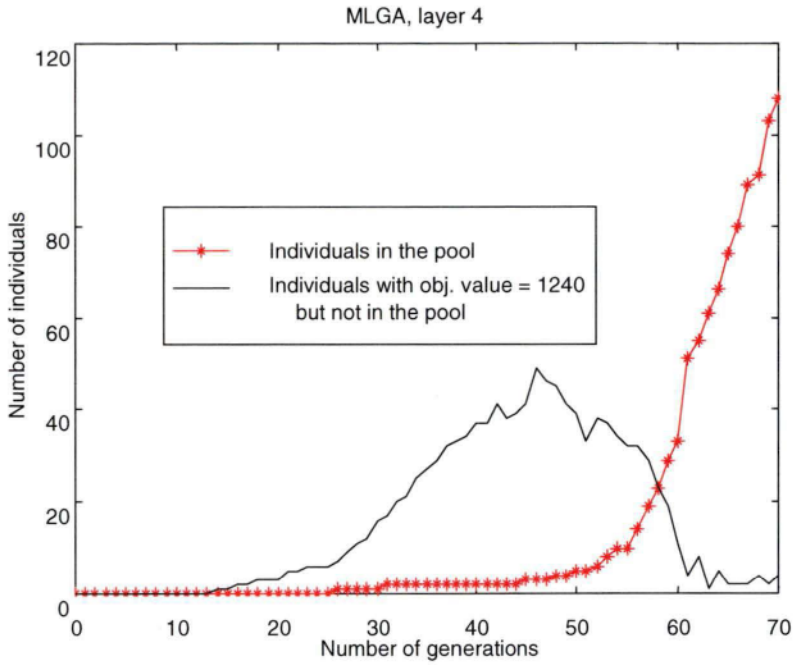

Figure 3.6 Growth of individuals in the pool during the first few generations

The first individual with a fitness equal to 1240 was found in the fourteenth generation, but it did not satisfy Criterion 3.2. It was preserved in the elite part of the population as the best individual so far, but was not included in the pool. During the first fifty generations the elite part of the population consisted mostly of such individuals, satisfying Criterion 3.1 but not Criterion 3.2.

The first sub-schedule suitable for the pool was found in the 26th generation. At first the number of such individuals grew very slowly, but after a while, since they were given preference to stay in the elite part, they completely forced all others out of the population. After 60 generations the number of individuals with high fitness but not complying with Criterion 3.2 is close to zero.

The fast growth of the individuals with high fitness but unsuitable for the next layer gene pool in the beginning of the run shows that if the pool criterion was not modified, half or more of the gene pool at the next layer would have been unusable. By using Criterion 3.2 however, the 4th GA layer was able to supply the next layer with sub-schedules of better quality.

3.5.4 *Evaluation of the 12-layer GA with modified pool criterion*

Tables A3.6 to A3.10 in Appendix 3 present layered results for the 12-layer MLGA's performance after Criterion 3.2 for the gene pool was implemented. The tables' structure and data entries are similar to the ones discussed in Section 3.4.2. As can be seen from the tables, the MLGA success rate was improved. When the retained reserve parameter R^0 was 1220, 1230 and 1240, the MLGA was able to find solutions in 37, 28 and 7 cases out of 40 instead of 34, 27 and 2 as in the earlier experiments.

After the pool criterion modification, the most evident improvement in MLGA performance was when the retained reserve parameter, R^0, was 1240. The comparison of the two types of pool criteria is presented in Table 3.5. The table contains data extracts from Tables A3.3 and A3.8, Appendix 3. It also includes an additional parameter for the MLGA with Criterion 3.2, this being the number of runs with the best objective value of at least 1240. With Criterion 3.1, all such solutions were included into the next layer gene pool and the layer was considered successful without an additional check. However, with Criterion 3.2, the existence of solutions with objective value equal to the pool threshold, R^0, is not a guarantee of a successful run since the solutions have to satisfy Eq. (3.7) as well to be included in the next layer gene pool.

Table 3.5 Performance of the 12-layer MLGA with different gene pool criteria

Layer	MLGA with Criterion 3.1		MLGA with Criterion 3.2		
	Number of successful runs	Number of generations	Number of successful runs	Number of runs with the best obj. value of 1240	Number of generations
1	40	57.05	40	40	58.78
2	40	182.55	40	40	110.50
3	40	43.85	40	40	45.90
4	40	123.80	40	40	105.23
5	36	199.90	40	40	194.80
6	21	187.92	31	31	181.28
7	18	149.29	26	28	151.42
8	11	183.72	14	20	177.73
9	2	216.36	7	7	180.00
10	2	33.50	7	7	12.86
11	2	48.50	7	7	73.29
12	2	16.00	7	7	11.29

As can be seen from Table 3.5, the 5th layer of the modified MLGA was successful in all 40 runs instead of the original 36 out of 40, which resulted in the next, 6th layer, being able to find solutions in 31 runs instead of only 21. Then layer No. 7 was able to find sub-schedules providing a reserve of 1240 MW 28 times, although twice these sub-schedules did not satisfy Criterion 3.2 and the runs were declared unsuccessful. According to Table 3.5, the 8th layer success rate does not improve much, with 11 versus 14 successful runs for the two gene pool criteria. However, another six runs of the 8th layer with Criterion 3.2 were declared unsuccessful even if a number of sub-schedules with the objective value of 1240 were found but did not satisfy Eq. (3.7) and as a result, the 9th layer was not run at all in these six runs. The higher quality of the genes in the 9th layer gene pool, by the MLGA with Criterion 3.2, resulted in the MLGA finishing successfully in 7 cases out of 14 instead of 2 cases out of 11, when the algorithm proceeded to the 9th layer. As in the previous group of experiments, the 9th layer proved to be the critical one and if some suitable sub-schedules were found the rest of the layers were also successful.

The modification of the gene pool criterion also reduced the number of generations in a run. Table 3.5 shows the number of generations taken over all runs of each layer. In most layers the algorithm ran for roughly the same number of generations, but in the 2nd and 9th layers, the MLGA with Criterion 3.2 was faster, with 111 generations instead of 183 in the 2nd layer, and 180 instead of 216 in the 9th layer. In layer No. 11, the number of generations in the second group of experiments was greater, 73 instead of 49, but since in the first group of experiments this layer was run only twice, the results are not conclusive.

The influence of Criterion 3.2 on the number of generations in a run is especially noticeable when R^0 is 1220 or 1230, as shown in Tables A3.6 and A3.7. For example, with R^0=1220; the slowest layers, No. 2, 6 and 9, benefited from the better quality gene pool, with the number of generations in successful runs decreasing from 97 to 68, from 93 to 77, and from 133 to 86 in the respective layers. When R^0=1230, the number of generations in successful runs decreased in the 2nd layer from 134 to 83, and from 151 to 81 in the 6th layer. On the other hand, an MLGA with Criterion 3.2 took more time to find solutions in the 8th layer, 41 generations instead of 33, but this still resulted in a faster search in the following layers.

The modified gene pool criterion helped to improve the quality of solutions found by MLGA layers and consequently, made it easier to find solutions in the next layers. However, since Criterion 3.2 only checks the units of the next layer, this does not guarantee that the units in subsequent layers, if there are any, will have enough available time slots to be scheduled. Moreover, the criterion does not necessarily provide room even for units in the next layer. For example, if there are two identical units in the next layer, when the corresponding sets of allowed values, $\mathcal{A}_j'$, are found, they may contain a single element, and as a result, the two units would be scheduled for maintenance in the same week. If the gross reserve in this week is not sufficient for both units to be stopped, then one unit will be impossible to schedule.

It could also be argued that Criterion 3.2 could be further enhanced to include checking of the units in subsequent layers, but apart from the inevitable increase in the time needed for the calculation of the fitness function, this improvement still would not guarantee sufficient accuracy in the evaluation of the sub-schedules, as the units from the next layers could only be checked for allowed numbers on an individual basis. Even if there are slots where each of the units could be scheduled, several of these units would be still competing with each other. In view of the diminishing benefits for increasing cost of processing time and more complex algorithm, Criterion 3.2 given in Eq. (3.7) has been left without change, comparatively simple and providing a certain degree of quality to the sub-schedules found for the next layer gene pool.

3.6 Second example of unit groupings

3.6.1 A 9-layer MLGA

While initial experimentation identified a 12-layer MLGA as the most effective, it is not the only possible way of dividing the units into groups for scheduling in consequent layers. An obvious alternative would be to reduce the number of layers. As was shown in Table 3.2, a GA is able to schedule more than fifteen units in the first layer. Thus in an MLGA with less layers the first layer could consist of 16 units with the capacity 210, 230 and 460 MW; that is, the same fifteen units that were used in the first layer of the 12-layer MLGA, plus unit No. 10. According to Table 3.2, this results in the number of

generations in the first layer run increasing by 1.5 to 2 times, depending on the retained reserve parameter, R^0. However, the addition of any further units was considered infeasible as the number of generations needed to fill the gene pool in the first layer escalates even further, also shown in Table 3.2.

According to Tables A3.1 to A3.5, the second layer in the 12-layer MLGA is one of the slowest. Moving unit No. 10 from it into the first layer can also result in a faster search in the second layer of the new MLGA as there are just two units left, units No. 18 and 19. From a number of experiments with these two layers it was found that the second layer finishes without any difficulty in about 10 generations, which makes it one of the fastest layers in the MLGA. However, in order to reduce the number of layers, it was decided to add another unit to the second layer.

As shown in Table 3.6, unit No.21 was added to the second layer of the new MLGA from the 4th layer of the 12-layer MLGA. It was selected since it not only has quite a large capacity, 100 MW, but also requires a long maintenance period, five weeks. In fact, if a product of the unit's capacity, C_j, by its maintenance requirement, M_j, is considered as the ordering factor, unit No. 21 should be the first of all remaining units. For example, units No. 1, 2 and 3, that were scheduled in the third layer of the 12-layer MLGA, have capacity of 150 MW and need three weeks of maintenance, therefore their capacity by maintenance product, $C_1 \times M_1 = 150 \times 3 = 450$ is less than that of unit No. 21, $C_{21} \times M_{21} = 100 \times 5 = 500$.

Table 3.6 First layers in an MLGA

12-layer MLGA	
Layer	Units
1	4 5 6 7 8 9 11 12 13 14 15 20 37 38 39 40
2	**10** 18 19
3	1 2 3
4	11 **21** 36

New MLGA	
Layer	Units
1	4 5 6 7 8 9 **10** 11 12 13 14 15 20 37 38 39 40
2	18 19 **21**
3	1 2 3
...	...

Another way to reduce the number of layers in an MLGA is to combine the easily scheduled units from the last few layers into a smaller number of layers. As shown in Tables A3.1 to A3.5, layers No. 10, 11 and 12 were always successful and, moreover, layers No. 10 and 12 were the fastest in the 12-layer MLGA. Thus layers No. 11 and 12

were combined into a single last layer, while unit No. 22 was added to the units scheduled in layer No. 10, as shown in Table 3.7.

Table 3.7 Last layers in an MLGA

12-layer MLGA	
Layer	Units
...	...
9	**22** 23
10	41 42 43
11	30 31 32
12	33 34 35

New MLGA	
Layer	Units
...	...
second last	**22** 41 42 43
last	30 31 32 33 34 35

Once the first two and the last two layers were determined, there are 14 units left to be scheduled. A number of different combinations of units were trialed. One of the most successful examples of unit grouping is presented in Table 3.8, together with the resulting number of genes and the length of genes in a chromosome (see Section 3.2.3). The 14 units are distributed among five layers, with the units' order being slightly different from the one in the 12-layer MLGA. The experimental results of the following 9-layer MLGA were reported in (Kelareva and Negnevitsky, 2001; Kelareva and Negnevitsky, 2002).

Table 3.8 9-layer MLGA

Layer	Units being scheduled	Number of genes	Gene length
1	4 5 6 7 8 9 10 12 13 14 15 20 37 38 39 40	16	1
2	18 19 21	4	1,1,1,16
3	1 2 3	4	1,1,1,19
4	16 17 28 29	5	1,1,1,1,22
5	11 36	3	1,1,26
6	23 24 25	4	1,1,1,28
7	26 27	3	1,1,31
8	22 41 42 43	5	1,1,1,1,33
9	30 31 32 33 34 35	7	1,1,1,1,1,1,37

3.6.2 Evaluation of the 9-layer MLGA

The results of the experiments with the 9-layer MLGA are given in Appendix 4. Tables A4.1 to A4.5 present the results for the MLGA with Criterion 3.1, while Tables A4.6 to

A4.10 contain the results of the experiments with Criterion 3.2. The structure of the tables is the same as in Tables A3.1 to A3.10 and data entries are an average of up to 40 runs.

Since the first layer now contains an additional unit, No. 10, which has a large capacity and a long maintenance period, it takes longer to find enough solutions to fill the gene pool for the second layer. For example, it took more than 70 generations instead of 50 for the MLGA with R^0=1220, as shown in Tables A4.1 and A3.1. On the other hand, replacing unit No. 10 with a 'simpler' unit No. 21 in the second layer resulted in a faster search in layer No. 2 of a 9-layer MLGA (43 generations instead of almost 100 in the same experiments). Thus, the first two layers in the 9-layer MLGA were able to schedule 19 units faster than the first two layers in the 12-layer MLGA scheduled 18 units.

As can be seen from Tables A4.1 to A4.10, the 9-layer MLGA was able to successfully schedule the first three layers for all values of R^0 except 1260. These three layers contained the twenty-two units with the largest capacity and the longest maintenance periods. The 9^{th} layer, now containing all the units from the 11^{th} and 12^{th} layers of the 12-layer MLGA, was also always successful. The 8^{th} layer, with the additional unit No.22, was also successful in all runs with the modified gene pool criteria. Evidently, unit No. 22 with the capacity of 50 MW and the maintenance requirements of 4 weeks presented more difficulty in scheduling than the other units in the last two layers. If the sub-schedules found by the 7^{th} layer could not provide unit No.22 with enough reserve for scheduling and therefore did not satisfy Criterion 3.2 in Eq. (3.7), the run was unsuccessful. The most difficult layers, according to Tables A4.1 to A4.10, proved to be 6^{th} and 7^{th}.

The best results were achieved, as expected, with Criterion 3.2. The success rate with R^0=1220 was 40 out of 40 (100 %), while with Criterion 3.1, the algorithm only succeeded in 36 runs out of 40 (90 %). With R^0=1230 the number of successful runs with Criterion 3.2 increased from 27 to 28. Table 3.9 shows the difference in performance of a 9-layer MLGA with both pool criteria when the retained reserve parameter is equal to 1240. Data in the table represents extracts from Tables A4.3 and A4.8, plus the number of runs when the best objective value found was 1240, as in Table 3.5.

Table 3.9 Performance of the 9-layer MLGA with different gene pool criteria

Layer	MLGA with Criterion 3.1		MLGA with Criterion 3.2		
	Number of successful runs	Number of generations	Number of successful runs	Number of runs with the best obj. value of 1240	Number of generations
1	40	86.68	40	40	82.05
2	40	76.58	40	40	65.30
3	40	83.03	40	40	68.05
4	31	213.28	28	35	200.83
5	19	199.55	24	24	170.71
6	10	202.47	14	14	176.00
7	3	185.20	8	10	175.07
8	2	114.00	8	8	48.13
9	2	125.50	8	8	42.88

According to Table 3.9, if the number of successful runs is considered, the MLGA with Criterion 3.1 seems to perform better, succeeding in 31 out of 40 runs in the third layer, while the algorithm with Criterion 3.2 finds good solutions only 28 times. However, the latter algorithm actually finds solutions with the objective value in 35 runs, but in 7 of them the solutions found do not satisfy Eq. (3.7) and the algorithm terminates. By providing consecutive layers with gene pools of better quality, the MLGA with Criterion 3.2 was able to find good schedules in 8 runs out of 40 (20 %), compared to the MLGA with Criterion 3.1 at 2 out of 40 (5 %).

As with the 12-layer MLGA, another advantage of using the modified pool criterion is the reduction in running time. Most layers ran for a similar or reduced number of generations (see tables in Appendix 4 and Table 3.9), and, besides, with the additional check of the solutions the algorithm did not run some layers, thus reducing the number of generations in unsuccessful runs.

3.6.3 Comparison of the 12-layer and 9-layer MLGAs

Table 3.10 summarises the results of MLGA performance with 12 and 9 layers. As can be seen from the table, there are many similarities in both MLGAs. For example, when the gene pool criterion is modified, the MLGA success rate improves, especially when R^0 =1240, while the number of generations decreases. Also the success rate of the two

types of MLGA is almost the same in corresponding experiments, especially with the retained reserves of 1230 and 1240.

Table 3.10 Summary of the MLGA performance

Number of layers in MLGA	Retained reserve parameter, R^0	Number of successful runs out of 40	Number of generations in a run
12-layer with Criterion 3.1	1220	34	656.65
	1230	27	853.95
	1240	2	1001.73
	1250	0	859.05
	1260	0	903.20
12-layer with Criterion 3.2	1220	37	510.63
	1230	28	763.75
	1240	7	1000.40
	1250	0	663.90
	1260	0	618.70
9-layer with Criterion 3.1	1220	36	627.73
	1230	27	766.89
	1240	2	771.50
	1250	0	799.20
	1260	0	723.20
9-layer with Criterion 3.2	1220	40	442.35
	1230	28	675.38
	1240	8	720.80
	1250	0	594.00
	1260	0	604.80

There are other common features in the performance of 12 and 9-layer MLGAs. If the tables from Appendices 3 and 4 are examined, it can be noticed that the 9-layer algorithm always successfully scheduled the 22 units from the first three layers, except for when R^0=1260. Similarly, the 12-layer MLGA was almost always successful in scheduling 24 units from the first four layers. These units are most difficult to schedule with the largest capacity and/or maintenance requirements. The last ten units in two layers in the 9-layer algorithm and the last nine units in three layers in 12-layer case were also scheduled easily. This leaves about ten units in the middle, scheduled in four or five layers, that present the main problem. These units are of medium capacity, require from 3 to 7 weeks for maintenance, and if they are successfully scheduled, the whole run is successful.

The similarities in MLGA behaviour suggests that MLGA success only partly depends on the number of layers. While the exact partitioning of a GA into layers has to be a

matter of trial and error, there is a general rule that can be suggested for layering the GA while solving this type of scheduling problem:

C×M rule Units are scheduled according to their capacity multiplied by the number of maintenance intervals needed, that is,

- Unit i is scheduled before unit j if $C_i \times M_i > C_j \times M_j$.

  However, this does not have to be a strict rule. For example, in the 12-layer MLGA units No. 16 and 17 were scheduled before units No. 28 and 29, even though the product of the unit's capacity by the number of maintenance intervals of the latter units is greater. When in a different unit grouping these two layers swapped places, the success rate did not change. Therefore, a more precise rule would be the following:

- Unit i is scheduled before unit j if $C_i \times M_i \gg C_j \times M_j$.

The only significant difference in MLGA performance was the smaller number of generations needed for the 9-layer algorithm, which indicates that an MLGA with too many layers may be unnecessary slow. A number of other unit groupings were tried and most of them performed similarly if the number of layers was from 9 to 12 and the order of the units in the layers was not considerably changed. That is, the success rate for a MLGA with Criterion 3.2 was 90 to 100 % for R^0=1220, 60 to 70 % for R^0=1230 and 15 to 20 % for R^0=1240. There were no successful runs with R^0>1240.

However, when the number of layers was less than 9 the success rate declined, especially for R^0=1240. Recall that a traditional GA, having just one layer, was unsuccessful (see Table 2.8). When the units were divided into two or three layers, even if the first one was successful, the overall rate of success was zero. Therefore, even though further reduction in the number of layers may lead to faster algorithm, it can also result in a lower success rate. Thus, some general rules concerning the number of layers in a MLGA can be formulated as following:

- Too many layers in an MLGA increase running time and may result in less efficiency (success rate of a 9-layer MLGA with R^0=1220 and modified pool criteria was 100 % compared to a 12-layer MLGA, at 93%).
- Too few layers in a MLGA may result in a layer that proves too hard to solve and as a result, the overall success rate may be reduced.

3.7 MLGA versus traditional GAs

Table 3.11 presents a comparison of an MLGA versus a traditional GA, and a GA with an indirect representation using a schedule builder. The MLGAs have 12 or 9 layers, and employ Criterion 3.2 for gene pools.

Table 3.11 Comparison of different algorithms performance
Number of successful runs out of 40

Retained reserve parameter, R^0	Traditional GA	MLGA		Schedule builder	
		12 layers	9 layers	'Deepest first'	'First available'
1220	0	37 (92.5%)	40 (100%)	33 (82.5%)	40 (100%)
1230	0	28 (70%)	28 (70%)	17 (42.5%)	40 (100%)
1240	0	7 (17.5%)	8 (20%)	0	37 (92.5%)
1250	0	0	0	0	0
1260	0	0	0	0	0

As can be seen, both MLGAs perform significantly better than the traditional GA with a direct representation. While the GA could not find any solutions of good quality, the new algorithm was able to build schedules providing a nett reserve of up to 1240 MW. Note also that the traditional GA was run for up to 5,000 generations, as shown in Table 2.8, while the number of generations in each layer of the MLGA was limited to 400 (see Section 3.3 4). The average number of generations in a run over a total of 40 runs was from between 440 to 1,000, as shown in Table 3.10.

An MLGA also compares favourably with the 'deepest first' schedule builder, discussed in Chapter 2. It can be argued that the schedule builder was running for a smaller number of generations, only 300, but, as can be seen in Table 2.5, the schedule builder is very slow in finding the first sequence that can be transferred into a good quality schedule. It takes about 180 generations to find the first individual providing the objective value of 1220, and almost 250 generations to find an individual with an objective value of 1230. At the same time, a MLGA is stopped after 200 generations if it fails to find any suitable sub-schedule in a layer, as explained in Section 3.3.4. Thus the MLGA performance can still be considered better than that of a single layer GA with an indirect performance and the 'deepest first' schedule builder.

Table 3.11 shows that a MLGA still does not perform as well as the 'first available' schedule builder. Only the 9-layer MLGA with modified pool criteria was able to find suitable schedules all the time with R^0=1220, although the number of generations required to obtain a good solution is still much larger than that of a schedule builder, 442 generations compared to 64 (see Tables 3.10 and 2.6). When R^0=1230 the success rate of an MLGA is no more than 70% and when R^0=1240, it is about 20%, while the corresponding success rate of the 'first available' schedule builder were 100% and 92.5%.

The inferiority of an MLGA compared to the 'first available' schedule builder can be explained by the large search space of the MLGA due to its use of a direct representation. When an MLGA finds a number of solutions suitable for the next layer gene pool, it effectively directs the next layer's search into particular areas of the problem domain. These areas are predefined by the sub-schedules found in the previous layer, and if the previous GA layer missed some potentially successful sub-schedules, they will never be recovered. Nevertheless, the MLGA technique of directing the search into different areas of the problem domain in different runs is dissimilar from the search by a GA with an indirect representation performed in the same area every run, restricted by the design of the schedule builder, as was illustrated in Figures 3.1 and 3.2.

The advantage of an MLGA is that in a successful run it finds about 300 completely different maintenance schedules, as shown in Appendixes 3 and 4, while a schedule builder often produces identical schedules from different individuals/sequences, as discussed in Chapter 2. Moreover, some of the schedules found by an MLGA may be beyond a schedule builder, since the latter explores only a part of the problem's domain.

3.8 Conclusion

In this chapter, a new multi-layered genetic algorithm (MLGA) was proposed and implemented. It utilises the idea of dividing a complex problem into simpler sub-problems. A solution-schedule is progressively built by dividing the power system's units into layers based on their capacity and maintenance requirements. Each GA layer builds a maintenance schedule from sub-schedules found in the previous layer by

adding new units to them. This way the algorithm searches through specific areas predefined by the results of the previous layer. The algorithm exploits the partial separability of the problem as well as the ability of a GA to produce not a single solution but a number of them. A few general guidelines were suggested for dividing the case study problem of maintenance scheduling into layers.

There are several essential features of the MLGA:

- The algorithm utilises a direct representation of a scheduling problem with a poolwise initialisation. One gene in an individual in all layers except the first represents a sub-schedule built in the previous layer. One or two-point crossover and poolwise mutation are employed.
- The MLGA uses the elitist replacement strategy with 10% of the population representing the elite at the beginning of a run. When the number of individuals suitable for the next layer gene pool is more than 10% of the population, the elite part is expanded to include all such individuals. At the end of the layer run all individuals suitable for the next layer gene pool are included into it.
- To maintain the diversity in the population the duplicates are weeded out.
- Interchangeable sub-schedules are allowed to stay in the population, but are deleted from the gene pool after being transferred to the next layer.
- Gene pools for the units scheduled in a particular layer are reduced before the layer starts if some units in the sub-schedule gene pool have converged.

This chapter also examined some aspects of the MLGA that improve the efficiency of the algorithm. These aspects include:

- The algorithm becomes faster and more efficient if Criterion 3.2 for the gene pool is employed to ensure the better quality of the sub-schedules that are passed onto subsequent layers. The success rate of the algorithm in some experiments increased by 300%, while the number of generations in a run decreased by 30% as the result of employment of Criterion 3.2.
- When the number of individuals suitable for the next layer gene pool is more than 70% of the population, a number of randomly built individuals are added in order to speed up the search for the last few members of the pool. Increasing the population by 7% allowed reducing the number of generations by 25% in some layers.

The new algorithm provides a better exploration of the problem domain when compared with the traditional GA that uses the same direct representation. While the latter failed to produce any schedules of good quality at all, the MLGA success rate ranged from 20 to 100%, depending on the retained reserve parameter. However, the MLGA performance is inferior when compared with the best of the GAs employing a schedule builder, which has a success rate of 90 to 100%, besides, being up to seven times faster.

In order to improve an MLGA's success rate, it would therefore appear necessary to further increase the efficiency of the MLGA search. There is certainly a potential for improvement, either by more thorough parameter tuning, or by coupling the MLGA with some other search method of local scope. To this end, an MLGA enhanced by a local search is examined in the next chapters.

CHAPTER 4

Multi-layered genetic local search

4.0 Introduction

This chapter examines the effect of incorporating a local search into the MLGA, which was introduced in Chapter 3. First, the notion of combining a local search technique with a GA is explained, and its implementation is examined when applied to a case study for the maintenance schedule optimisation. Next, the new enhanced algorithm, a multi-layered genetic local search (MLGLS) is defined and its parameters are discussed and experimentally tuned. Finally, the MLGLS is trialed on various unit groupings, including those examined in Chapter 3.

4.1 Genetic local search (GLS)

As was mentioned in Chapter 1, an effective way to improve the performance of a GA is to combine it with another optimisation method suited to the domain of the particular problem. Every optimisation technique can be characterised by advantages and disadvantages (Davis, 1985; Michalewicz, 1999). If it is possible to combine the most favourable features of the different techniques, the resulting algorithm may be able to produce better synergistic solutions than those obtainable using the individual techniques by themselves (Davis, 1985; Kelly and Davis, 1991).

For example, GAs represent a global optimisation method but often lack the ability to perform an effective and fast local improvement once a number of sub-optimal solutions are found, while many classical hill-climbing techniques can be efficient local optimisers (Powell et al, 1991; Mansour and Fox, 1991). Thus the global performance of a GA could be enhanced by combining it with a local search technique (LS) with the resulting combination often called a *genetic local search* (GLS) (Ibaraki, 1997). As the GA and the LS characteristics complement each other, the drawbacks of both techniques can be counterbalanced, that is, the GA ability to search for optima in a large problem domain is balanced with the LS efficiency in improving an initial solution (Braun and Zagorski, 1994). Therefore the typical GA fault of not being able to refine a solution is minimised by using the LS. At the same time, the use of an LS within a GA framework prevents the LS from convergence to a local optimum in a problem domain with widely spaced multiple optima (Powell et al, 1991; Ibaraki, 1997).

When performing an LS within a GA, various optimisation techniques can be used, depending on the particular problem. For example, Braun and Zagorski (1994) examined a hybridisation of a GA (the authors called it an evolutionary algorithm because the only genetic operator used in this implementation was mutation) with a gradient descent technique for evolving and training neural networks. This way the evolution of a topology of a network was enhanced by a local weight optimisation traditionally used for network training. The resulting networks were compared to the ones obtained by evolutionary algorithms only, and/or by training predesigned neural networks. According to Braun and Zagorski (1994), the hybrid algorithm managed to simplify the topology of the networks by discarding unnecessary weights and/or inputs while keeping the same standard of performance, or even improving it.

Depending on the choice of basic theory incorporated in a GLS, that is, Darwinian model versus Lamarckian model, GLS techniques can differ in their approach to using the results of the LS within the GA. When a *Darwinian* approach is used, the local improvements are not recorded into an individual's genotype because, according to Darwin's evolution theory, the genotype does not change during an individual's life and therefore the features acquired during a local optimisation cannot be passed to the successive generations. In this approach the local optimisation is used mostly as the means to evaluate an individual according to its ability to improve (Whitley et al, 1994; Renders and Flasse, 1996; Quagliarella and Vicini, 1997).

In contrast, the *Lamarckian* approach suggests that the acquired features and skills can be passed to offspring by reproduction. Although Lamarckian principles are considered to be wrong by the accepted evolution theory, they can be utilised when artificial evolution, such as GAs, is considered. As the latter approach was found to be more successful (Grefenstette, 1991; Renders and Flasse, 1996), it is the Lamarckian model that is used in this thesis.

Sometimes a local optimisation is just an extension of a mutation operator used on each new individual with probability 1 and with a variety of possible improvements considered in one iteration, as was implemented by Whitley et al in (1994) for function optimisation with binary encoding. The authors used an LS which changed individuals by flipping one binary gene in them; and if the resulting solution had better fitness than the current one, the current individual was replaced by the best neighbour.

Thus, a GLS can be realised by introducing an additional operator performing a local optimisation without changing the GA structure. Usually, the *neighbourhood* of an individual solution *s*, denoted as $\mathcal{N}(s)$, is defined as a set of individuals from the search space obtained from *s* by changing its components in some specified way. In the above example such a neighbourhood contained all individuals obtained from the current one by changing the value of exactly one gene (Whitley et al, 1994).

A typical GLS has an additional operation in every generation (Ibaraki, 1997). That is, every individual's neighbourhood is searched and a better individual replaces the initial one, i.e., the initial individual's genotype is changed according to the Lamarckian approach. The new individual can be the first one found with a fitness better than the initial one, in such a case the GLS is called *first*. Alternatively, the best individual in the entire neighbourhood can replace the initial one, and the GLS is then called *best* (Ibaraki, 1997). An example of the pseudo-code of a GLS is given in Figure 4.1

```
begin GLS
        t = 0
        initialise P(t)
        F(t) = evaluate P(t)
        until (terminating condition) do
                t = t + 1
                P'(t-1) = select_for_reproduction(P(t-1))
                P'(t) = recombine_and_mutate(P'(t-1))
                for every p ∈ P'(t)
                        search_neighbourhood(p)
                        if (∃ p' | F(p') > F(p)), replace(p by p')
                end % for loop
                P(t) = replace(P(t-1) by P'(t))
                evaluate P(t)
        end
end
```

Figure 4.1 Pseudo-code for a GLS

Improvement by an LS can take place at different stages of a GLS. For example, instead of locally improving the offspring population, as shown in Figure 4.1, it is possible to perform an LS on the parent population before selection for reproduction takes place (Ibaraki, 1997).

4.2 Multi-layered GLS for a maintenance scheduling problem

The case study problem deals with the maintenance schedule optimisation in a power generating system. A direct representation of the problem was chosen since it allows the algorithm to explore the problem's entire domain, as explained in Chapter 2. However, this representation resulted in a search space too large to be efficiently examined by a traditional GA. Because of this, a new algorithm, an MLGA, was suggested and implemented in Chapter 3. This algorithm divides the units into layers and builds the maintenance schedules gradually, using partial schedules found in previous layers as separate long genes. The aim of a GA in each layer is to find a number of such sub-schedules to be passed onto the next layer. Criterion 3.2 is used to control the quality of the sub-schedules in the next layer gene pool.

An elite replacement strategy is used in an MLGA with 10% of the population representing the elite at the beginning of a run. When the number of individuals suitable for the next layer gene pool is more than 10% of the population, the elite part is expanded to include all such individuals. At the end of the layer run all individuals suitable for the next layer gene pool are included into the elite. To maintain diversity in the population, duplicates are weeded out.

Although the MLGA approach proved to be more effective than the traditional GA examined in Chapter 2, it still lacked the ability to produce solutions of a good quality every time. To improve the algorithm success rate, an LS algorithm can be added to it, and in accordance with the convention identifying a GA combined with LS as a GLS, a multi-layered GA with an LS can be called a multi-layered GLS or an MLGLS.

When an LS operator is added into an MLGA algorithm, each layer represents a separate GLS, similar to the one in Figure 4.1. The aim of such a GLS is, as in an MLGA, to provide the next layer with a gene pool of good quality sub-schedules that can be built up into larger schedules by adding new units to them. Therefore, the LS algorithm is used only to speed up the search for the suitable sub-schedules.

Since the aim of a GLS in each layer is to find a number of sub-schedules satisfying Criterion 3.2, an LS is applied only to the individuals that do not satisfy this criterion and need improvement.

4.2.1 *Definition of a neighbourhood*

There are several issues that should be examined when implementing an LS as an additional operator in an MLGA as proposed in Chapter 3. Firstly, the neighbourhood of an individual/schedule that is considered in the LS should be suitable for direct representation of the case study problem. As previously explained, an individual represents a schedule with genes denoting the start of a maintenance outage of a corresponding unit. The neighbourhood of an individual should describe a group of individuals with properties similar to those of the initial individual. In the case of a maintenance schedule, it would be, for example, all schedules that assign the corresponding units for maintenance in almost the same time slots.

General definition of a neighbourhood of a schedule

In order to specify a neighbourhood of a schedule formally, we define a distance between two individuals in the population, which can be used as a measure of closeness between individuals. For example, a Euclidean distance between two points on a plane, $X1$ and $X2$ with coordinates $(1,1)$ and $(4,3)$, is calculated as follows (Burden and Faires, 1989):

$$|X1-X2|_{Euclidean}=\sqrt{(4-1)^2+(3-1)^2}=\sqrt{13}=3.61.$$

However, in this research the distance between two points (or two individuals/schedules) is defined as the maximum difference between the coordinates (or values of corresponding genes), which is denoted sometimes as $|\ |_\infty$ (Burden and Faires, 1989),

$$|X1-X2|_\infty=\max\{|4-1|,|3-1|\}=3.$$

Such distance is easier calculated and more suitable for the integer-valued individuals.

Therefore, the neighbourhood of a schedule represents a hypercube enclosing a traditional Euclidean neighbourhood, as shown in Figure 4.2 for a two-dimensional case. As can be seen from the figure, $X1$ does not belong to the Euclidean neighbourhood of $X2$ with radius 3 but is included in the enclosed neighbourhood based on the maximal difference.

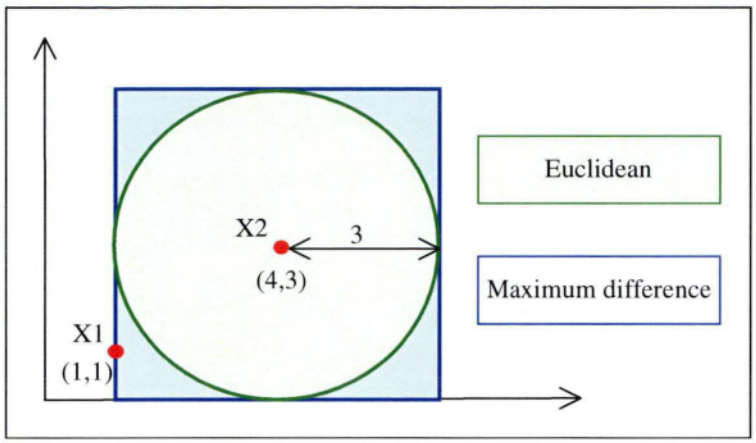

Figure 4.2 Eucledean and maximum difference neighbourhoods with radius 3

For example, suppose a schedule s allocates four units for maintenance, starting with weeks 12, 13, 20 and 40, that is, $s=(12,13,20,40)$. Then a schedule $s'=(11,14,19,40)$ is in the neighbourhood of s with radius 1, since the gene values in s' differ from the corresponding gene values in s by no more than 1. Thus, a neighbourhood of a schedule that assigns maintenance time to N units can be defined as follows:

Definition 4.1 Let an individual $s=(a_1,a_2,\ldots,a_N)$ be a potential solution to a maintenance scheduling problem with a gene value a_j taken from the corresponding gene pool $\mathcal{A}_j$ and defining the beginning of a scheduling outage for the unit j. Then if r is a given positive number, the neighbourhood of s with radius r is the set of all individuals $s'=(a'_1,a'_2,\ldots,a'_N)$, such that $a'_j \in \mathcal{A}_j$ for all $j=1,\ldots,N$ and

$$|a_j - a'_j| \le r \text{ for all } j=1,\ldots,N. \tag{4.1}$$

The neighbourhood of s with radius r is denoted as $\mathcal{N}_r(s)$.

Definition 4.1 is essentially a geometrical definition of a neighbourhood that represents a hypercube around the individual. The side length of the hypercube is $2r$, where r is the radius of a neighbourhood, and all individuals inside the hypercube belong to that neighbourhood. This definition is perfectly suitable for many problems such as the optimisation of a real-argument function, especially if gene values are continuous and compact (without 'holes').

However, in the case of a scheduling problem with a gene taking its value from a discrete gene pool like the pools shown in Appendix 1, Definition 4.1 does not always

serve the purpose of finding individuals with similar features. For example, consider the neighbourhood of a single gene (or the neighbourhood of an individual consisting of only one gene). Suppose that the gene, a, has a current value of 5, while the corresponding gene pool contains the following eleven values:

$$\mathcal{A} = \{1,2,3,5,6,10,25,26,27,28,49\}.$$

Then, as shown in Figure 4.3.a, if radius $r = 2$, the neighbourhood of $a = 5$ contains only two elements other than 5, $\mathcal{N}_2(a) = \{3,5,6\}$, that is, the available values for the gene are 3, 5 and 6, since only they answer the Eq. (4.1). If the gene has a different value, for example, $a = 10$, the neighbourhood of the gene is empty if the radius is less than 4.

It seems reasonable to define the neighbourhood in such a way that would allow a gene to have some neighbours even if its value is isolated in a gene pool, such as 10 and 49 in the gene pool $\mathcal{A}$. This becomes possible if the neighbourhood of a schedule includes individuals with the corresponding gene values r **positions** apart in the gene pool, rather than r being the numerical difference between the gene values. This way, for example, the neighbourhood of gene $a = 5$ with radius 2 includes five elements, $\mathcal{N}_2(5) = \{2,3,5,6,10\}$, as shown in Figure 4.3.b; and for gene with the value 10, the corresponding neighbourhood is $\mathcal{N}_2(10) = \{5,6,10,25,26\}$.

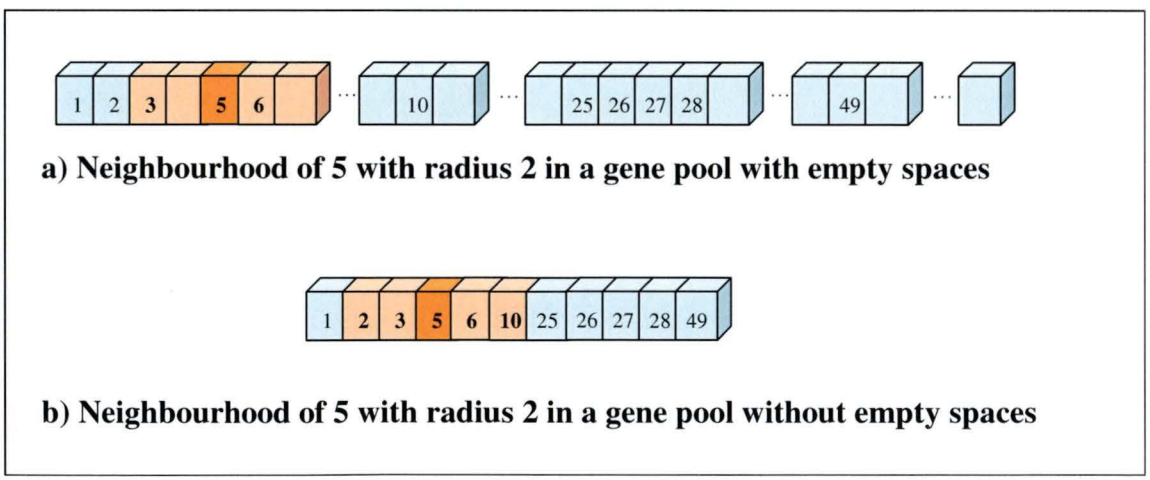

Figure 4.3 Gene pools with and without empty spaces

Next, when the neighbourhood of 49 with $r = 2$ is considered, it now includes the neighbouring values of 27 and 28. However, as the value 49 is the last one in the gene

pool, it does not have any neighbours on its right, which reduces the number of possible changes that can be made to the gene while performing a local search. Also, the value 49 would be selected less frequently than a value in the middle of the gene pool, since only the values on the left from 49 will include it in their neighbourhoods. Therefore, to get a more even search, it makes sense to think of the gene pool as a circle with the 'ends', that is, values 49 and 1, adjacent to each other. In this case, the neighbourhood of 49 with radius 2 includes 27 and 28 as well as 1 and 2, as shown in Figure 4.4.

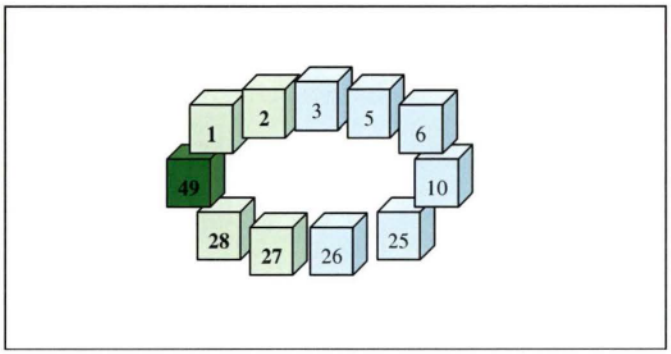

Figure 4.4 Neighbourhood in a circular gene pool

Thus the previous definition of a neighbourhood is changed into the following:

Definition 4.2 Let an individual $s=(a_1,a_2,\ldots,a_m)$ be a potential solution to a maintenance scheduling problem with a_j defining the beginning of a scheduling outage for the unit j, taken from the corresponding gene pool $\mathcal{A}_j=\{a_{j1},a_{j2},\ldots,a_{jL}\}$ containing L elements with the cyclic order as follows:

$$a_{j1}\prec a_{j2}\prec\ldots\prec a_{jL}\prec a_{j1},$$

and the distance between two elements in $\mathcal{A}_j$ defined as the minimal number of elements between them plus one. Then an individual $s'=(a_1',a_2',\ldots,a_m')$ belongs to $\mathcal{N}_r(s)$, if $a_j'\in\mathcal{A}_j$ for all $j=1,\ldots,m$ and the relative distance between the two values is less than or equal to r. That is, provided that a_j is in l^{th} position in the gene pool and a_j' is in k^{th} position,

$$\min\left(|l-k|,L-|l-k|\right)\le r. \tag{4.2}$$

4.2.2 *Definition of a neighbourhood made suitable for an MLGA layer*

Definitions 4.1 and 4.2 are applicable to a complete schedule, or a sub-schedule with every gene of length 1 taking its value from a corresponding gene pool. However, this definition does not cater for some problem-specific requirements particular to an MLGA. For example, in all MLGA layers k, for $k>1$ each chromosome $s^{(k)}=\left(a_1,\ldots,a_{N^{(k)}},s^{(k-1)}\right)$ contains $N^{(k)}$ genes that correspond to a single unit each; plus one long gene $s^{(k-1)}$, representing a sub-schedule of $N^{(1,\ldots,k-1)}$ units. The value of this long gene is taken from a special gene pool $\mathcal{S}^{(k-1)}$ that holds all sub-schedules found in the previous layer. If Definition 4.2 is used to find a neighbourhood of an individual $s^{(k)}$ with a 'long' gene, this gene $s^{(k-1)}$ has to be treated in the same way as the rest of the chromosome $s^{(k)}$, and the units in $s^{(k-1)}$ may be rescheduled in a different way. This may result in a new sub-schedule $s'^{(k-1)}$ that is not included into the gene pool $\mathcal{S}^{(k-1)}$ and, while such a deviation may provide an additional exploration of the scheduling problem domain, it may also result in illegal sub-schedules of the units contained in $s^{(k-1)}$. Since a considerable amount of effort has been made to build legal sub-schedules in the previous layers, it would be unreasonable to examine illegal sub-schedules instead of the legal ones provided in $\mathcal{S}^{(k-1)}$.

An alternative way of finding the neighbours of an individual $s^{(k)}$ could be by finding sub-schedules from the gene pool $\mathcal{S}^{(k-1)}$ which are similar to $s^{(k-1)}$, but the question remains of how to measure the degree of similarity of the two sub-schedules. To calculate the distance between every two members of $\mathcal{S}^{(k-1)}$ would take a considerable amount of time if the pool is large. Therefore it was suggested that in MLGA layers after the first, the neighbourhood of an individual includes only individuals with the same long gene and is defined as following:

Definition 4.3 Let an individual $s^{(k)}=\left(a_1,a_2,\ldots,a_{N^{(k)}},s^{(k-1)}\right)$ in layer $k>1$ be a sub-schedule where a_j, defining the beginning of a scheduling outage for the unit j, are taken from the corresponding gene pool $\mathcal{A}_j$; and $s^{(k-1)}\in\mathcal{S}^{(k-1)}$ is a sub-schedule found in the previous layer. Then if r is a given positive number, the neighbourhood

$\mathcal{N}_r\left(s^{(k)}\right)$ of $s^{(k)}$ with the radius r is the set of all individuals containing the same sub-schedule $s^{(k-1)}$, $s'^{(k)} = \left(a'_1, a'_2, \ldots, a'_{N^{(k)}}, s^{(k-1)}\right)$, such that

$$a'_j \in \mathcal{A}_j \cap \mathcal{N}_r\left(a_j\right) \text{ for all } j = 1, \ldots, N \tag{4.3}$$

where the neighbourhood of a_j is defined according to Definition 4.2.

4.2.3 *Evaluation of a neighbourhood*

Evaluation of the neighbours can be significantly simplified when Definition 4.3 is used. LS methods are known to be a considerable slow-down for an algorithm since they evaluate a large number of additional individuals, and evaluation is usually the most time-consuming procedure in a GA (Ibaraki, 1997). However, if the additional individuals are obtained according to Definition 4.3, evaluation can be performed much faster by calculating the nett reserve provided by the long gene at the beginning of evaluation, and then recalculating the difference after adding the units from the current layer.

As was discussed in Chapter 3, the nett reserve R_i in week i provided by an individual in layer k of an MLGA is calculated according to Eq. (3.3), that is,

$$R_i = S - P_i - \sum_{j \in J_i^{(1,\ldots,k)}} C_j = G_i - \sum_{j \in J_i^{(1,\ldots,k)}} C_j,$$

where S is the system's capacity, P_i is the predicted load in week i, G_i is the gross reserve of the system in week i, C_j is the capacity of a unit j and $J_i^{(1,\ldots,k)}$ contains the units stopped for maintenance in week i from the current layer k as well as from the previous layers.

Instead of evaluating each neighbour separately and, for this reason, completely recalculating the nett reserve provided by each neighbour, the evaluation procedure can be divided into two steps. Suppose, for example, that the neighbourhood of an individual $s^{(k)} = \left(a_1, \ldots, a_{N^{(k)}}, s^{(k-1)}\right)$ consists of M neighbours, each neighbour described as $s_m^{(k)} = \left(a_{1m}, \ldots, a_{N^{(k)}m}, s^{(k-1)}\right)$, for $m = 1, \ldots, M$, where the long gene $s^{(k-1)}$ is

the same as in $s^{(k)}$ and gene values a_{jm} comply with Definition 4.3. First, the nett reserve provided by the common long gene $s^{(k-1)}$ is calculated for each week i as following:

$$R_i^{(k-1)} = G_i - \sum_{j \in J_i^{(1,\ldots,k-1)}} C_j, \tag{4.4}$$

where C_j is the capacity of a unit j, and $J_i^{(1,\ldots,k-1)}$ are the units contained in the sub-schedule $s^{(k-1)}$ and scheduled for maintenance in week i. Then, for each neighbour m, $m = 1,\ldots,M$, the corresponding nett reserve in week i is calculated as:

$$R_{im} = R_i^{(k-1)} - \sum_{j \in J_{im}^{(k)}} C_j, \tag{4.5}$$

where the set $J_{im}^{(k)}$ includes the units scheduled in week i according to schedule m, but not the units contained in the common long gene $s^{(k-1)}$.

This way, instead of calculating the sum of the capacities of all units scheduled in the layers from the 1st to k^{th} for each neighbour m, only the capacities of $N^{(k)}$ units scheduled in layer k are subtracted from the precalculated nett reserve of the sub-schedule $s^{(k-1)}$. To further optimise evaluation of the neighbours, the nett reserve of each gene $s^{(k-1)}$ from the gene pool $S^{(k-1)}$ can be calculated in the beginning of the layer and stored as an attribute of this gene.

4.2.4 *Gene pool modification for neighbourhood exploration*

Another advantage of using Definition 4.3 to explore the neighbourhood in MLGA layers other than the first, is the opportunity of excluding some unsuitable gene values from the gene pool of the single changing genes. This procedure employs the same idea that was utilised in modification of the gene pools for single genes in a chromosome in case of unit convergence (see Section 3.3.3). The long gene plays the role of an additional load on the system, and the gene pools for the changing genes are modified accordingly.

For example, if the neighbourhood of an individual $s^{(k)} = \left(a_1, \ldots, a_{N^{(k)}}, s^{(k-1)}\right)$ is examined, it is unnecessary to consider all possible gene values for each a_j, but only the values that do not violate the problem constraints given in Eq. (2.6). First, the nett reserve in week i provided by the sub-schedule $s^{(k-1)}$, $R_i^{(k-1)}$, is obtained with the help of Eq. (4.4). It is then used as the gross reserve of the system, G_i, and according to it, new sets of allowed values for the single genes are calculated, taking into account the retained reserve parameter R^0, as was done in Eq. (2.14). Thus, a gene pool $\mathcal{A}_j$ for unit j is defined as follows:

$$\mathcal{A}_j = \{1, \ldots, 52 - M_j + 1\} \cap \{i \in \{1, \ldots, 52\} \mid R_i^{(k-1)} - R^0 \geq C_j\}, \tag{4.6}$$

where M_j is the number of weeks required for maintenance of unit j and C_j is the capacity of the unit.

Note that since all long genes that have been included in the gene pool are assessed by Criterion 3.2, a gene pool $\mathcal{A}_j$ for unit j is never empty even if the long gene $s^{(k-1)}$ is considered as an additional load on the system.

4.2.5 *Choosing the GLS parameters*

Radius and number of changing genes

Once the neighbourhood of a schedule/sub-schedule is defined, it is possible to examine other aspects of an LS algorithm when applied to the case study problem. The radius of a neighbourhood, r, is an obvious parameter for the LS, determining the size of the neighbourhood to be explored. Another parameter relevant to the neighbourhood size is the number of the genes in a chromosome allowed to change their values at one time. According to the definitions discussed above, any number of genes in an individual may change within the range defined by the radius of a neighbourhood. Nevertheless, it is clear that two neighbourhoods with the same radius will be different if the amount of genes allowed to change while producing neighbours varies.

For example, if one neighbourhood is formed around an individual by changing only one gene at a time, the changes are limited, the neighbourhood is small and, therefore, the search is slow and may fail to find a better solution. On the other hand, if all genes change values every time a new neighbour is produced, it results in a larger neighbourhood, and the algorithm may have a better chance of exploring the problem's search space. However, a large number of changes may cause a significant disruption in an individual and the resulting neighbours will differ greatly from the initial sub-schedule, which may help or hinder the improvement of an individual. Therefore the number of changing genes, Nc, is a meaningful LS parameter; and together with the radius of a neighbourhood, r, it can cover a variety of different changes in an LS algorithm.

If, for example, an individual s has Ng genes of length 1, exactly Nc of which change, the number of combinations of Ng elements taken Nc at a time is

$$^{Ng}C_{Nc} = Ng!/Nc!(Ng-Nc)! \tag{4.7}$$

Every changing gene can take $2r$ values different from its current value, provided that all gene pools have at least $2r+1$ elements. Therefore, the number of neighbours with exactly Nc genes changed, is calculated as the number of combinations, $^{Ng}C_{Nc}$, multiplied by $(2r)^{Nc}$.

However, by definition, a gene value can change in the range $[-r,r]$ including zero change and, in fact, the initial individual itself belongs to its neighbourhood as well and should be included in the count. Thus, the total number of individuals in the neighbourhood with Nc genes changing in the range $[-r,r]$, denoted as $\left|\mathcal{N}_r^{Nc}(s)\right|$, is calculated as the sum of the sizes of neighbourhoods with exactly i genes changed, where i takes values from 0 to Nc:

$$\left|\mathcal{N}_r^{Nc}(s)\right| = \sum_{i=0}^{Nc}(2r)^{Nc-i}\, Ng!/(Nc-i)!(Ng-Nc+i)! \tag{4.8}$$

In the experiments described below, the number of changing genes, Nc, was allowed to have one more value, *. In this case the neighbourhood is formed by first choosing a random number of genes, m, for each individual, from 1 to the total number of genes of length 1, and then changing the values of only m genes in the individual. Obviously, if

Nc =*, the size of the neighbourhood is the same as in case of $Nc = Ng$, the largest neighbourhood possible with a given radius r:

$$\left|\mathcal{N}_r^*(s)\right| = \left|\mathcal{N}_r^{Ng}(s)\right| = \sum_{i=0}^{Ng} (2r)^{Ng-i}\, Ng!/i!(Ng-i)! \tag{4.9}$$

However, with large values of Ng and r equal to the new value, *, the actual number of changing genes is evenly distributed within the range from 1 to Ng, which allows a different type of a neighbourhood exploration when only a small number of neighbours is considered. In contrast, when $Nc = Ng$, mostly the individuals with the actual number of changing genes close to Ng are considered. The procedure of finding a random neighbour of an individual s is given in Figure 4.5. It shows the difference in the algorithm when Nc is an integer or *.

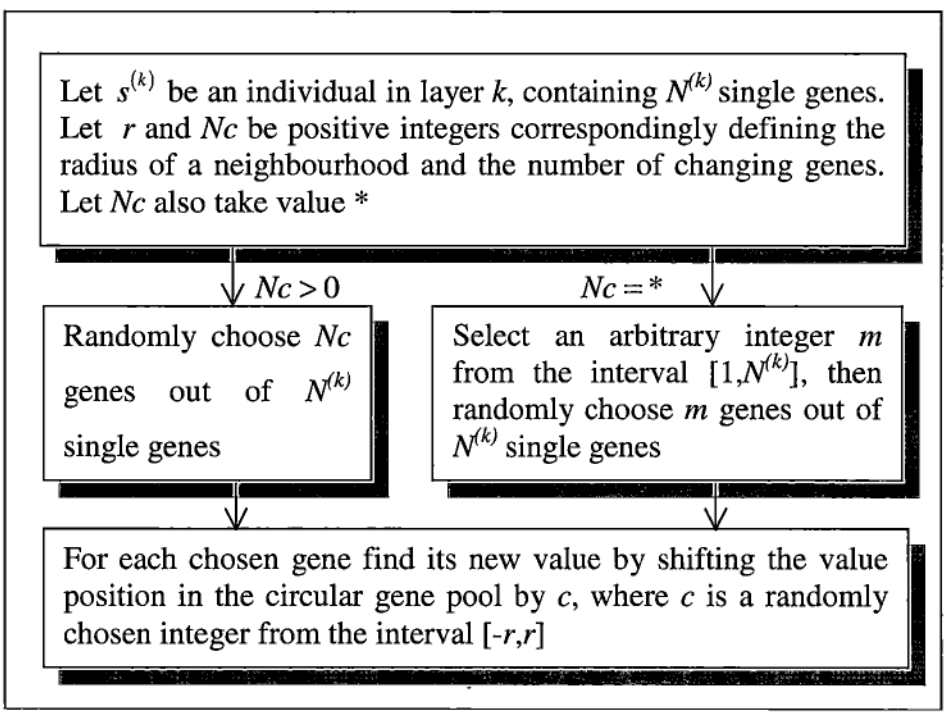

Figure 4.5 Neighbourhood definition with various *Nc*

Table 4.1 presents some examples of the neighbourhood size with different radius and number of changing genes in an initial individual containing ten single genes. The resulting neighbourhood properties are also summarised in the table.

Table 4.1 Neighbourhood size and properties depending on *r* and *Nc*

Radius, *r*	Number of changing genes, *Nc*	Size of the neighbourhood of an individual with 10 genes	Neighbourhood properties
1	1	21	Small neighbourhood, may not include any improvements
5	1	101	Slightly larger neighbourhood, still may not include any significant improvements
1	5	12,585	A variety of changes in a large neighbourhood. Better chance of finding an improvement
5	5	27,424,601	Very large neighbourhood, includes highly disrupted versions of the initial schedule
1	10	59,049	The largest possible neighbourhood with a fixed radius, includes all possible combinations of changes
5	10	2.59E+10	

Number of neighbours sampled

The intensity of a neighbourhood examination is another aspect of an LS algorithm. If a neighbourhood is small, it can be thoroughly searched reasonably fast. On the other hand, while it is highly possible to find a good improvement in a large number of individuals, it would take a considerable amount of time to check all the schedules in a vast neighbourhood with a large radius and a substantial number of changing genes. Therefore, another parameter of an LS algorithm is the number of neighbours sampled when the neighbourhood of an individual is examined, denoted here as *Ns*. This parameter should change according to the neighbourhood size; and the efficiency of an LS search depends on it. It should be noted again that one of the most time consuming procedures in many GA problems is fitness evaluation (Ibaraki, 1997). When a GLS is performed on every member of the population of size, say 300 individuals, with each neighbourhood consisting of 100 neighbours, then 30,000 individuals will be evaluated during every generation, not counting the chromosomes obtained by recombination and mutation. This may lead to an unacceptable increase in running time.

Thus, the number of individuals to be sampled from the neighbourhood, *Ns*, is an important LS parameter. If *Ns* is large, it may lead to positive changes in a few generations, but it may take a lot of time evaluating the neighbours, especially in the first layer of an MLGLS (see Section 4.2.2). Conversely, when the neighbourhood is large but sampled sparsely, the LS may overlook some good neighbours.

4.3 Tuning the GLS parameters

A series of experiments were conducted to find some optimal GLS parameters, that is, the radius of a neighbourhood, r, the number of changing genes in a chromosome, Nc, and the number of neighbours sampled when performing the neighbourhood search, Ns.

Two types of the algorithm were examined, with the LS being performed after and before genetic operators, and these algorithms are denoted as GLSa and GLSb respectively. Recall that a GLSa algorithm was presented in Figure 4.1, with an LS performed on the population of offspring. The GLSb algorithm differs from GLSa in that an LS is performed on the population of parents before genetic operators take place.

The experiments described in Chapter 3 showed that an MLGA algorithm can be quite efficient if the retained reserve parameter, R^0, was 1220 or 1230. The corresponding success rate was up to 100% and 70% (see Table 3.10). However, when R^0=1240, the success rate was no more than 20%. At the same time it was unclear whether it is possible to obtain schedules with the retained reserve larger than 1240 MW. For these reasons, the GLS parameters were tried on the case study problem with R^0 equal to 1240.

4.3.1 Layering units for an MLGLS

In Chapter 3 two MLGAs, consisting of 12 and 9 layers, were examined. When a MLGLS was tried on the same layers of units, the LS accelerated the search in the first layers to such an extent that it became clear that the algorithm can be successful even with more units in fewer layers. Therefore, although MLGLS with the same unit layering, as in Chapter 3 are examined later in this chapter, it was decided to choose a different way to divide the units into layers for the examination of GLS performance with varying parameters. As was discussed in Chapter 3, after the units are ordered according to the CxM rule, they can be divided roughly into three groups depending on their difficulty:

1. The first 22 or 24 units with the largest capacity by maintenance value that are always successfully scheduled by an MLGA in 3 or 4 layers.
2. The next 10 or 11 units that proved to be difficult to schedule once the units from the first group are allocated the best time slots.
3. A group of 9 or 10 units that are always successfully scheduled in two or three layers, provided the previous groups of units are scheduled.

Keeping the above information in mind, the units are divided into layers as shown in Table 4.2.

Table 4.2 Unit groupings for an MLGLS

MLGLS layer	Units being scheduled	Number of genes	Gene length
1	4 5 6 7 8 9 10 12 13 14 15 18 19 20 37 38 39 40	18	1
2	1 2 3 21 28 29	7	1,1,1,1,1,1,18
3	11 16 17 36	5	1,1,1,1,24
4	23 24 25	4	1,1,1,28
5	22 26 27	4	1,1,1,31
6	41 42 43	4	1,1,1,34
7	30 31 32 33 34 35	7	1,1,1,1,1,1,37

The 18 units scheduled in the first layer were chosen because when they were scheduled in one layer by a GA, it was a rather slow process, as was shown in Table 3.2. For that reason the first layer in an MLGA contained no more than 16 units (see Sections 3.3.1 and 3.6.2). For the MLGLS it was hypothesised that the additional LS operation might improve the algorithm's performance sufficiently for the 18 units to be scheduled in just one layer.

The second layer includes six units, which is double the number of the second layers in Chapter 3. This is done because, when the second layer contained only three units, a GLS would usually finish the layer in two to four generations (see a further description of experiments with 12 and 9-layer MLGLS), meaning that the layer was too easy for it. The experiments described in this section were set to examine a GLS performance in conditions that are relatively hard.

Thus, the first 24 units are scheduled in just two layers. The following ten units of medium capacity and maintenance requirements are then scheduled in three layers, which are followed by the final two layers with the easiest units.

4.3.2 *Possible parameters values*

The experiments examined GLS behaviour on each of the three first layers, with the GLS parameters taking a variety of values, as shown in Table 4.3. The two first layers contain 24 easily scheduled units and, from previous experiments, a GLS should be expected to succeed every time in both layers. However, the optimal GLS parameters can be quite different in these layers, because the neighbourhood of an individual is defined differently in the first and subsequent layers. Apart from this, GLS performance may differ even in layers with the same number of new units being scheduled, since the percentage of potentially good solutions in a population may recede in the difficult layers and grow again in the last, easy layers. Because of this variety of conditions, experiments were conducted to determine the optimal parameters for different layers. The entries in Table 4.3 show the values of the GLS parameter being examined in the experiments. As can be seen from the table, r and Ns took the same or almost the same values in all three MLGLS layers, unlike Nc, which takes different values depending on the layer, for the reasons explained below.

Table 4.3 GLS parameters

MLGA layer	Number of new units scheduled	Radius	Number of changing genes	Number of neighbours
1	18	1 2 3 5 10	* 1 3 5 10	10 50 100
2	6	1 2 3 5 10	* 1 2 3 5	50 100
3	4	1 2 3 5 10	* 1 2 3	50 100

The first layer schedules the eighteen units with the largest capacities and maintenance periods, and theoretically, all eighteen genes in an individual can change their values simultaneously within the range allowed by the radius of the neighbourhood. However, this would hardly be advisable since it inevitably would result in a substantial disruption of the sub-schedule. Therefore, it was decided that the number of changing genes in an individual should not be too large, taking values of 1, 3, 5 and 10. Also Nc can be assigned value * which would mean that each neighbour will be obtained by changing a randomly selected number of genes, from 1 to 18, according to the algorithm presented

in Figure 4.5. This way the neighbourhood will contain a large variety of different neighbours, as was explained in Section 4.2.2.

The second layer, as shown in Table 4.2, consists of only six units, therefore, the number of changing genes cannot be greater than six since another gene of length 18, representing a sub-schedule built in the previous layer, does not change during a neighbourhood search. Apart from the value *, the number of changing genes in an individual takes values of 1, 2, 3 and 5. For the same reason the number of changing genes in the third layer is *, 1, 2 and 3.

It should be noted that although the number of neighbours sampled is shown in Table 4.3 as 10, 50 or 100; in reality it would be unnecessary to sample a small neighbourhood with $Ns = 100$. For example, a neighbourhood of an individual in the third layer with 4 genes of length 1 contains only 9 neighbours if $r = 1$ and $Nc = 1$, as follows from Eq. (4.8). In such a case the number of neighbours sampled, Ns, is reduced accordingly and, if the size of the neighbourhood is less than 50, $Ns = 100$ is not used at all.

4.3.3 Gene pools and other MLGLS parameters

In the experiments that try to determine the optimal GLS parameters, the second layer GLS was taking the values for the long gene from a pool filled with sub-schedules found in the first layer. This gene pool is the same in all experiments with the second layer.

A GLS in the third layer also takes values for the long gene from a gene pool filled with sub-schedules found in the previous layer, but these do not necessarily resemble real-world conditions. As further shown, while the third layer almost always successfully schedules the units, it does not produce the same number of solutions all the time. Moreover, the quality of these solutions can vary, depending on many factors, including the initial solutions found in the first layer GLS in a run. At the same time, it is necessary to examine the influence of different parameters on GLS performance with the same starting conditions. Therefore, these conditions were deliberately made relatively easy. The gene pool for the third layer is filled with solutions found in the

second layers of two different runs (200 genes were selected from one run and 100 from another), in both runs the third layer was able to find about 250 solutions.

It is also necessary to comment about other parameters of the algorithm, such as genetic operators and their probabilities. Recombination is performed by a two-point crossover if the number of genes of length 1 is greater than 4, and by a one-point crossover if otherwise. Crossover probability is taken to be equal to 1, since an elite approach is implemented with the initial generation gap of 10% and increasing according to the number of individuals suitable for the next layer gene pool.

It may seem that mutation becomes unnecessary since its role of a random change of a gene is taken over by the LS. However, an LS in the neighbourhood of an individual with a long gene is performed without changing this gene and, therefore, mutation retains its importance as a random change of a long gene in a chromosome. While the mutation rate can be assigned to 0 in the first layer, in all consecutive layers it remains a means to explore the search space more thoroughly, and in certain situations can be increased from its initial rate of 0.01, as will be explained further.

Similarly to the experiments in the previous chapters, duplicates are weeded out of the population and replaced by new randomly built individuals. On the other hand, interchangeable sub-schedules are allowed to stay in the population but are discarded from the gene pool in the next layer (see Section 3.3.2). The termination condition for the experiments concerned with the GLS parameters, was that the algorithm is run for a maximum of 40 generations, unless the next layer pool was filled earlier.

4.3.4 Experimental results

All experiments discussed in this section were run twenty times. The results of the experiments are presented in Appendix 5. Tables A5.1 to A5.3 show the results for the first layer GLS parameters, Tables A5.4 and A5.5 contain the results for the second layer while the results for the third layer are given in Tables A5.6 and A5.7. The data in the tables is sorted in descending order, using the column containing the number of distinct solutions found that are suitable for the next layer gene pool. Apart from the

GLS parameters, i.e., radius, number of neighbours sampled and number of changing genes; other entries in the tables are:

- Number of generations in a run.
- Best objective value at the end of the run.
- Number of genes in the next layer gene pool, that is, the number of non-interchangeable solutions found in the layer. This number can be less than the population size for two reasons: the algorithm was running for 40 generations already and failed to find more solutions; or there were too many interchangeable sub-schedules in the population (see Section 3.3.2).
- Generation when the first individual suitable for the next layer gene pool was found.

First layer GLS

As can be seen from the first three tables in Appendix 5, a thorough examination of possible GLS parameters for the first layer was conducted. In these experiments a GLS was initially run with a population of 200. The population size was taken to be less than the usual 300 only because evaluation of the individuals takes a significant time in the first layer, and adding an LS operator results in a further increase to this time.

In the first group of experiments a GLSa algorithm was run with all possible parameter variations. There are several facts that can be observed from Table A5.1:

- A neighbourhood of a medium size, that is, with small radius and a relatively large number of changing genes, provides the best opportunity to find an improvement for an individual. For example, in the best five entries of the table the radius had values of 1 and 2 twice, and 3 once; while the number of changing genes was 5, three times; and 3, twice. These combinations of parameters allowed the algorithm to find an average of 174 to 185 solutions in about 36 generations.
- The least successful combination of r and Nc seems to be when both of them take large values, like 5, and especially 10, for the radius; and likewise for the number of changing genes. The results of the experiments with these parameters are shown in the last quarter of the table and, as can be seen, the GLS was able to find less than 15 solutions in 40 generations.

- $Ns = 10$ is definitely not enough to find an improvement in a neighbourhood, since the best result produced with such a sampling rate was only about 100 solutions in 40 generations, which is almost twice less than the best number of solutions found by examining 100 and 50 neighbours, 185 and 176 correspondingly. As a result, it was decided not to use $Ns = 10$ in the subsequent experiments.
- On the other hand, the algorithm with $Ns = 50$ and $Ns = 100$ performed in a similar way. Among the top ten results in the table three are obtained with $Ns = 50$, the number of generations in the corresponding runs is also practically the same, and the number of solutions found differs by only about ten. Since the evaluation is very slow in the first layer, it seems unreasonable to spend extra time evaluating an additional 50 neighbours for every individual, just to find 10 more solutions in the same number of generations.

In the next group of experiments, presented in Table A5.2, the same algorithm, GLSa is tried on a population of 100 individuals. The results of these experiments can be summarised as following:

- A GLSa with a population of 100 at its best takes almost the same number of generations, about 31, to find the required number of solutions, 100; while it takes about 37 generations to find 185 solutions with a population of 200. Even if the initial population of a larger size will take more time to evaluate, by the end of the run only a small number of individuals have their neighbourhoods searched and, therefore, the total amount of evaluations is the same in both cases.
- As in the previous experiments, the best results are obtained with a radius of no more than 5, with a relatively small number of changing genes, from 1 (with radius 5) to 5 (with radius 1). The worst results are when both r and Nc are large.
- Unlike the previous experiments, the sampling size of 100 seems to be more successful than 50, although the difference in the number of solutions found still differs for approximately 10, and the number of generations required is almost the same.

After comparison of the performance of a GLSa with a different population size, it was decided that the population of 200 is more effective in the long run, since it provides almost twice as many solutions for the next layer gene pool, while taking just a few generations longer.

In the third group of experiments with the first layer, the performance of a different algorithm, a GLSb is examined. Recall that in this algorithm an LS is performed on the population of parents before genetic operators take place. The algorithm was run with a population size of 200, and the results of these experiments are given in Table A5.3.

To compare the two algorithms, GLSa and GLSb, the best five results from Tables A5.1 and A5.3 are combined in Table 4.4.

Table 4.4 GLSa and GLSb best results in the first layer

Algorithm	Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Number of genes in the next layer gene pool
GLSa	1	100	5	36.80	185.40
	2	100	3	37.30	183.10
	2	100	5	36.80	179.50
	1	50	5	36.65	175.95
	3	100	3	35.45	173.80
GLSb	2	100	3	39.45	162.20
	5	100	1	39.95	149.90
	10	100	1	39.95	142.30
	3	100	1	39.75	133.55
	1	100	5	39.55	132.00

There are a few observations that can be made when comparing the results presented in the table:

- The GLSb algorithm does not provide the same type of steady performance as the GLSa. For example, in the number of solutions found in a run between the first and fifth best, the results differ by about 12 in the case of GLSa, and by 40 in the GLSb. Moreover, the best result produced by a GLSb is just over 160 solutions compared to 185 by the GLSa.
- While the best result by a GLSb is obtained with the same parameters as the second best result by a GLSa, when $r = 2$ and $Nc = 3$; a GLSb obtains the next best results when only one gene was changing within the range determined by a larger radius, from 3 to 10.
- The sampling size of 100 proved to make a noticeable difference in the GLSb algorithm when compared to $Ns = 50$. In fact, as shown in Table A5.3, the top ten results were obtained with $Ns = 100$, and the difference in the performance depending on the sampling size is up to 50 solutions.

Therefore, the following conclusions can be made from the experiments concerned with the first layer GLS:

1. The GLSa algorithm proves to be more effective than the GLSb, especially since it allows the sampling size to be reduced to 50 instead of 100 without significant loss of efficiency
2. A sampling size of 10 is ineffective and is not used in further experiments unless stated otherwise.
3. The best values for the radius are 1 or 2, while for the number of changing genes the best values are 3 or 5.
4. Reduction in population size from 200 to 100 does not bring any significant improvement except, perhaps, some acceleration in the beginning of the run. The disadvantage is that a lower number of solutions is obtained during the same number of generations

Second layer GLS

Experiments for the second layer were conducted using the same two types of algorithm, GLSa and GLSb, with the same parameters. The results are presented in Tables A5.4 and A5.5. The best five results from these tables are combined in Table 4.5.

Table 4.5 GLSa and GLSb best results in the second layer

Algorithm	Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Number of genes in the next layer gene pool
GLSa	5	100	5	15.70	279.30
	5	50	5	20.90	275.35
	3	100	5	14.35	273.95
	10	100	5	15.80	272.35
	3	50	5	18.05	270.45
GLSb	5	100	5	15.60	281.40
	10	50	5	22.70	278.80
	3	100	5	15.20	278.20
	5	50	5	20.65	278.15
	10	100	5	16.20	276.35

The following conclusions can be made after examining Tables 4.5, A5.4 and A5.5:

1. GLSa and GLSb algorithms show a remarkable similarity in their performance with the number of solutions found being almost the same during a similar number of generations, if the GLS parameters had the same values.

2. The optimal number of changing genes is 5, as can be seen from Table 4.5. The radius can be taken to be 5 as well since in both algorithms its results are in the top five, no matter whether the sampling size was 50 or 100.
3. According to Tables A5.4 and A5.5, the least effective number of changing genes and radius value are both equal to 1.
4. The difference in performance produced by the sampling size can be noticed only in the number of generations needed to reach the required number of good solutions. An algorithm with Ns=50 took about 5 generations longer than the one with Ns=100, that is, about 20 instead of 15, if the top results are considered. Since in the layers other than the first the evaluation of the neighbours is performed faster (see Section 4.2.1) using Eq. (4.4) and (4.5), reduction in the number of generations by about 25% can be considered as an advantage and, therefore, Ns=100 is used.

Third layer GLS

The results presented in Tables A5.6 and A5.7 describe the performance of GLSa and GLSb in the third layer. Table 4.6 presents the best five results from both tables.

Table 4.6 GLSa and GLSb best results in the third layer

Algorithm	Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Number of genes in the next layer gene pool
GLSa	10	100	3	10.00	236.40
	3	100	*	11.05	235.65
	10	100	*	10.15	235.35
	5	50	3	10.85	234.80
	5	100	3	10.10	234.50
GLSb	10	100	*	11.65	238.15
	10	100	3	11.35	236.05
	5	100	*	12.00	235.65
	5	100	3	12.00	234.65
	10	50	3	12.35	233.60

After analysing the data in Tables 4.6, A5.6 and A5.7, it is clear that the GLS performance in the second and third layers has been very similar:

1. The GLSa and GLSb performed equally well, with the top ten results in both cases being from 228 to 238 solutions found during 10 to 14 generations. There is even less difference in the top five results.

2. As can be seen from Table 4.6, the best results were obtained when the number of changing genes was 3 or *, which ensured a variety of possible changes in a neighbourhood. The optimal radius values are also quite large, 5 and 10.

3. The least effective values of r and Nc were 1, as in the second layer. Evidently, the resulting neighbourhood is too small to guarantee an improvement in an individual.

4. The sampling size, Ns, did not make any noticeable difference in the algorithm performance, neither in the number of solutions found, nor in the number of generations the algorithm ran. Therefore, to save the computational time, the smaller value of Ns=50 is used unless stated otherwise.

Fourth and fifth layer GLS

Individuals in the fourth and fifth layers have a similar structure to the ones in the previous, third, layer, in that there is a small number of genes of length 1 and one large gene, taken from a gene pool provided by the previous layer GLS. Therefore, the following can be assumed:

1. GLSa and GLSb can be expected to perform similarly in these layers.

2. Small values for the radius and the number of changing genes will result in a small neighbourhood which may fail to provide an improvement of an individual.

5. The radius value should be large, 5 to 10. However, the gene pools for the changing genes are adjusted by using Eq. (4.6) every time a neighbourhood of a particular individual is searched. This may result in the gene pool containing only a small number of allowed values. If the modified gene pool contains no more than 10 values, neighbourhoods with radii 5 and 10 will contain exactly the same elements due to the circular nature of the gene pool (see Figure 4.4). r=5 is therefore a suitable value for the radius.

3. The number of changing genes should be either * or 3, to cover as many changes as possible.

4. Although the sampling size did not make any difference in the third layer, it may be more important in further layers, which are usually harder to schedule. Therefore, the number of neighbours sampled, Ns, should be kept equal to 100, especially if

Nc=*. Alternatively, Ns can be reduced if a special procedure keeps track on the number of allowed values in gene pools $\mathcal{A}_j$ when a neighbourhood of an individual is examined. Then the neighbourhood size is calculated using a modification of Eq. (4.8), which adjusts Ns accordingly.

The experiments with the fourth and fifth layer parameters confirmed these expectations.

Sixth and seventh layers GLS

The last two layers contain the units that are always easily scheduled, provided that the previous layers were successful, as was shown in Chapter 3. The sixth and seventh layer structure is similar to the third and second layer correspondingly and, therefore, it can be expected that a GLS will be successful, provided that the neighbourhood of an individual contains a sufficient number of various changes. This can be achieved if the radius is taken to be equal to 5, while the number of changing genes is providing a variety of changes, being either 3 or * for the sixth layer and 5 for the seventh layer.

4.3.5 Recommended GLS parameters

As a result of the experiments discussed above, the GLS parameter values that are recommended are shown in Table 4.7.

Table 4.7 Optimal GLS parameters

MLGLS layer	Population size	GLS type	Radius	Number of changing genes	Number of neighbours sampled
1	200	GLSa	1	5	50
2	300	GLSa / GLSb	5	5	100
3	300	GLSa / GLSb	5 / 10	3 / *	50
4	300	GLSa / GLSb	5	3 / *	100
5	300	GLSa / GLSb	5	3 / *	100
6	300	GLSa / GLSb	5	3 / *	50 / 100
7	300	GLSa / GLSb	5	5	50 / 100

4.4 MLGLS performance

4.4.1 MLGLS implementation

The MLGLS algorithm was implemented using the MATLAB programming language. The implementation incorporated a number of additional procedures used to define and search the neighbourhood of an individual, as described earlier in this chapter. Then an LS operator was added to an MLGA algorithm.

Several unit-layering schemes were tried, including the 7-layer MLGLS suggested in Table 4.2, as well as the 9 and 12-layer MLGLS with layering given in Tables 3.8 and 3.3 and discussed in detail in Chapter 3. The results of the experiments with the retained reserve parameter, R^0, equal to 1240 are presented in Appendix 6. The structure of the tables in Appendix 6 is the same as described in Section 3.4.2, when the layered results were examined in Chapter 3.

In all experiments both variations of the algorithm, MLGLSa and MLGLSb were tried. However, a GLSa was always employed in the first layer since it proved to be more effective (see Table 4.5).

The number of generations in each layer run was restricted to 80, with the termination criteria variables, $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$, being 20, 5 and 5 respectively. That is, starting at the 20^{th} generation, the GLS performance was evaluated every 5 generations, and the algorithm was stopped if the increase in the number of individuals suitable for the next layer gene pool during the last 5 generation was less than 10%. If the gene pool was empty, the run was declared unsuccessful. For more information on termination criteria see Sections 3.3.4 and 3.4.1.

Another specific detail of an MLGLS that should be mentioned is the population size in the first layer. As was shown earlier, an LS evaluation in the first layer takes a lot of time, especially in the beginning of the layer run, when almost all individuals get their neighbourhood searched for improvements. For this reason the initial population size in the first layer is taken to be 200. However, in order to provide more solutions for the next layer gene pool, the population is increased to 300 when the number of good solutions is close to 200. Randomly built new individuals are added and the algorithm runs for another few generations until the number of good solutions is over 300 or the

other termination criteria are satisfied. Note that the population is increased up to 320 at the end of every layer, to speed up the search (see Section 3.3.6).

4.4.2 *Layered results for the MLGLS*

7-layer MLGLS

The 7-layer MLGLS was run with GLS parameters as recommended in Table 4.7. Both variations, MLGLSa and MLGLSb, were tried with the results given in Tables A6.1 and A6.2 respectively. As can be seen from the tables, both algorithms were successful in 11 runs out of 40, which means a 27.5% success rate. This is 7.5% better than the best results achieved by the 9-layer MLGA (see Chapter 3). Moreover, an MLGLS runs for fewer generations than an MLGA: about 120 instead of over 700.

When the two variations, MLGLSa and MLGLSb are considered, the algorithms behave very similarly. Both algorithms were always successful in the first three layers, except one run of the MLGLSa; while the most difficult layers were the fourth and fifth with about one third of the fourth layer algorithm and about half of the fifth layer algorithm unsuccessful. A comparatively small number of solutions for the next layer gene pool were found in these layers (no more than 60 on average). The slowest layers in the 7-layer MLGLS were the first (over 40 generations on average) and the third (about 30 generations), while the fastest layers were the last two, taking usually only one to three generations to complete.

As can be seen from Table A6.2, an MLGLSb was unsuccessful in two runs in the last, seventh layer, after successfully finishing the previous layers. Table 4.8 presents the layered data from one of these unsuccessful runs. As can be seen from the table, the first two layers have found about 300 genes for the next layers, but in the subsequent layers the search is not as successful and, additionally, a large number of the solutions found turned out to be interchangeable. For example, in the third layer after 35 generations, 137 individuals in the population were suitable for the next layer gene pool, but only 70 of them represented unique sub-schedules, as was found when the good individuals were transferred into the next layer gene pool. Recall that two units scheduled in this layer, No. 16 and 17, have the same capacity and maintenance requirements, and thus

the reduction in the gene pool size is easily explained: almost every elite individual in the population had an interchangeable sub-schedule also in the population. While the interchangeable sub-schedules were allowed to remain during the layer run, almost half of them were discarded from the gene pool (see Section 3.3.2).

Table 4.8 Layered data from an MLGLSb run

Layer	Number of generations	First generation		Last generation		Number of genes in the next layer pool	Generation of the first 'good' individual
		Best obj. value	Number of 'good' individuals	Best obj. value	Number of 'good' individuals		
1	54	1060	0	1260	301	288	5
2	5	1180	0	1250	303	302	2
3	35	1180	0	1240	137	70	2
4	20	1220	0	1240	36	6	2
5	20	1230	0	1240	8	4	2
6	1	1240	62	1240	300	170	1
7	20	1220	0	1230	0	0	0

A similar situation existed in the next, fourth layer. In 20 generations, 36 candidates into the gene pool were found but only 6 of them were used in the next layer. The three identical units scheduled in the fourth layer can be rearranged in 6 different ways and, evidently, the algorithm found all these combinations.

In the fifth layer, the search over 20 generations found 8 individuals suitable for the next layer gene pool, but, as two units scheduled in the layer were identical, the number of non-interchangeable sub-schedules was only 4. Nevertheless, the sixth layer produced 62 good individuals in the initial population, scheduling three units No. 41, 42 and 43 with relatively large capacities, but needing only one week for maintenance. Moreover, an LS which was performed on the rest of the population before the genetic operators took place was able to improve all the individuals so that they were considered suitable for the next layer gene pool. The sixth layer was completed in just one generation, and 170 non-interchangeable sub-schedules were found.

Unfortunately, all sub-schedules in the gene pool proved to be unusable in the last seventh layer. The question arises as to whether the search in the sixth layer was efficient enough, or if the algorithm missed sub-schedules that would have been able to accommodate the units from the last layer. In fact, when this particular run was examined, it was found that the four long genes in the gene pool were unusable and the

algorithm failed in scheduling the remaining nine units when the last two layers were combined into one.

Thus, the only possible reason for the run failure is the poor quality of the four sub-schedules found in the fifth layer, in that they could not provide enough of a reserve for scheduling all units in the last two layers. Recall, that Criterion 3.2 does not automatically guarantee that the subsequent units will be scheduled (see Sections 3.5.3 and 3.5.4). Since the search in layers 3 to 5 was conducted during at least 20 generations, it can be assumed that the search space in these layers was explored thoroughly. In fact, since the gene pools in the fourth and fifth layers were quite small, containing 70 and 6 long genes respectively, it is possible that all good solutions were found in the first few generations. This then raises a question about the quality of the solutions found in the first two layers. Although they both produced about 300 good sub-schedules, it is quite possible that most of them did not have the ability to accommodate the rest of the units.

Evidently, it is important to encourage a more thorough search in the first layers, however, simply discarding interchangeable individuals from the population on a regular basis would not necessarily help. If no sub-schedules were found in the first layers to build into a successful schedule through the subsequent layers, the whole algorithm is bound to fail.

Another conclusion that can be drawn from the experiments with the 7-layer MLGLS is that the last two layers can be combined into one without performance deteriorating. In fact, a number of experiments were conducted with such 6-layer MLGLS and the results were practically the same as for the 7-layer MLGLS

9 and 12-layer MLGLS

In Chapter 3 two MLGAs with different unit groupings were examined. The units were divided into 12 or 9 layers, as was shown in Tables 3.3 and 3.8. An MLGLS was tried on the same groupings of units, and the layered results with the retained reserve parameter, R^0 equal to 1240 are presented in Tables A6.3 to A6.6. The GLS parameters were chosen based on the experimental results described in Section 4.2.5 with some minor adjustments. For example, in the second layer, which contained only three units,

the number of changing genes was three instead of five. At the same time, in the layers containing only two units the radius was increased to 10, which provided up to 441 neighbours when both single genes were allowed to change, according to Eq. (4.9). By contrast, the neighbourhood of an individual with only two single genes and radius 5 contained just 121 neighbours.

As shown in Tables A6.3 and A6.4, the 12-layer MLGLSa and MLGLSb were successful only in three and two runs out of forty, the success rate being 7.5% and 5% respectively. This is worse than the 12-layer MLGA which was successful in 17.5% (see Table 3.10) and significantly worse than the 9-layer MLGLSa and MLGLSb, which succeeded in 12 and 10 runs respectively (30 and 25%) according to Tables A6.5 and A6.6. The poor performance of the 12-layer MLGLS can be postulated as follows:

- A small number of units in most layers minimises the effect of Criterion 3.2, which assesses the candidates into the gene pool according to their ability to provide enough reserve for the units in the subsequent layer.
- The above point becomes more relevant when coupled with the MLGLS ability to easily find all interchangeable units, which was evident in the example with the 7-layer MLGLSb in Table 4.8. Only two units were scheduled in each of layers 5 to 9 and in layers 5 to 8 the units are identical. This results in a large number of interchangeable sub-schedules in the population and, therefore, a reduced number of unique genes of good quality in the next layer gene pool.

Consequently, while all MLGLS algorithms were able to schedule the first 26 units successfully, the number of sub-schedules found by a 12-layer algorithm in five layers is less than that of a 9-layer algorithm found in four layers, that is, less than 40 compared to over 120 on average. The 7-layer algorithm does not have a layer in which schedules with 26 units are built. However, in its third layer it finds an average of over 70 sub-schedules with 28 units, which is half as much as the 9-layer MLGLS, but twice as much as the 12-layer algorithm. Evidently, about 30 sub-schedules with 28 units was not enough for a good performance of the 12-layer MLGLS. Note that the 12-layer MLGA was able to find an average of 100 solutions with 26 units, and about 75 with 28 units, according to Table A3.8.

Clearly the new algorithm with relatively high selective pressure of R^0=1240 works better when the number of layers is reasonably small, and the search is not too fast.

4.4.3 *MLGLS performance with various retained reserves*

Table 4.9 summarises the MLGLS performance with different unit layering and various retained reserve parameter values. While both variations of the algorithm were tried, the table presents the results for MLGLSa only, since the results of MLGLSb were very similar. In addition it includes the results of experiments with a single layer GLS, that is, a traditional GA which was enhanced with an LS operator in each generation. The GLS was run with various retained reserve parameters and the table shows the range of the best objective value found after 300 generations in 40 runs.

As can be seen from the table, all algorithms were quite successful with R^0 less than 1240. Both MLGA and MLGLS had the success rate of 100% with R^0=1220, however, the MLGLS displayed the success rate of over 90% with R^0=1230, which is better than the MLGA that had the success rate of 70%. Also the MLGLS was about five times faster, needing 80-120 generations to find 300 solutions, compared to about 500 or 700 generations needed by the MLGA, according to Table 3.10. When R^0=1240, the 9-layer MLGLS has a success rate of up to 30%, which is an improvement on the 9-layer MLGA (20%). Moreover, the number of generations in a run is reduced to about 120 from the 720 generations needed by the MLGA.

Table 4.9 MLGLS performance with different number of layers

Retained reserve, R^0	12-layer MLGLS		9-layer MLGLS		7-layer MLGLS		1-layer GLS, best obj. value after 300 generations
	Successful runs out of 40	Number of generations in a run	Successful runs out of 40	Number of generations in a run	Successful runs out of 40	Number of generations in a run	
1220	40 (100%)	65.42	40 (100%)	74.63	40 (100%)	62.10	1170-1210
1230	37 (92.5%)	109.93	37 (92.5%)	116.34	39 (97.5%)	83.70	1180-1210
1240	3 (7.5%)	111.73	12 (30%)	113.25	11 (27.5%)	113.25	1170-1210
1250	0	111.50	0	116.48	0	106.80	1170-1220
1260	0	104.70	0	94.05	0	90.70	1170-1210

However, a 12-layer MLGLS was less successful than a 12-layer MLGA, with the success rate dropping by more than half, from 17.5% to just 7.5% when R^0=1240. The reasons for this were discussed in the previous section.

The new, 7-layer grouping proved to be quite efficient, having a success rate similar to the 9-layer MLGLS, and finding the 300 solutions slightly faster when R^0 is 1220 or 1230. All MLGLS algorithms failed to find solutions with a minimal retained reserve of 1250 or 1260 MW.

A single layer GLS, as can be seen from Table 4.9, could not compete with MLGLS algorithms. After 300 generations the maximal objective value obtained was, at best, 1210 MW, and only once in all 200 runs (40 runs for each R^0 value) a schedule with a minimal reserve of 1220 MW was built.

Note that each fitness evaluation in a single layer GLS takes longer than in an MLGLS. For example, even in the first layer of an MLGLS, only 15 to 18 single units are scheduled instead of 43, as in a single-layer GLS, which means that less than half of the number of operations is needed to perform the evaluation. Moreover, the fitness is evaluated even faster in subsequent layers, as discussed in Section 4.2.2. Therefore an examination of a neighbourhood is more than twice as slow in a single layer algorithm. This shows that layering the GLS algorithm can significantly accelerate the search, while simultaneously improving its performance.

4.5 Conclusion

This chapter examined the effect of combining a local search (LS) with an MLGA. First, several definitions of a neighbourhood were proposed, and the definition most suitable for the direct representation of a scheduling problem was selected. Next MLGLS parameters were examined and tuned via a series of experiments. As a result, the following rules for choosing MLGLS parameters can be recommended:

- In the first layer an LS should be applied to the population after genetic operators to speed up the search and increase the success rate. In subsequent layers an LS operator can be employed either before or after genetic operators.
- In the first layer of an MLGLS, the radius of a neighbourhood should be small while the number of changing genes should be between one third and one fourth of the chromosome length.

- In the subsequent layers, all single length genes should be allowed to change, while the radius of the neighbourhood should be large, especially if the number of single genes is small.
- To prevent unnecessary increase in computation time, the number of neighbours sampled in the first layer should be kept small.
- In the subsequent layers, the number of neighbours sampled depends on the difficulty of the layer, that is, the number of generations that the layer takes to complete. The difficulty of a layer depends on the proportion of suitable solutions in the layer search space, which can be estimated from the previous experiments and from the size of the gene pools in the layer.

The experiments with the MLGLS algorithm show that including an additional LS operator into an MLGA significantly improves its performance, especially with 7 and 9 layers, and the success rate increased by 30 to 50%, depending on the retained reserve parameter value. On the contrary, a 12-layer MLGLS was less successful than a 12-layer MLGA with a large retained reserve parameter, because of the MLGLS ability to fill the population with a large number of interchangeable solutions. However, even when the success rate remained about the same, an MLGLS requires at least five times fewer generations to complete the run.

Adding an LS operator to any GA always increases the number of fitness evaluations and can slow down the algorithm. However, in an MLGLS the process can be sped up more than twice compared with a traditional GLS, due to the reduced number of genes in a chromosome. At the same time each generation in subsequent MLGLS layers takes only slightly longer than a generation in an MLGA, if the method proposed in this chapter is used for neighbourhood evaluation. Thus, the resulting algorithm is not only more successful but also faster than a basic MLGA, or a single layer traditional GLS.

Still, an MLGLS has the potential for further improvement. Several problems have been identified in MLGLS, including:

- An MLGLS easily finds all interchangeable solutions and a large proportion of the elite population have to be discarded when transferred into the next layer gene pool. As a result, gene pools in difficult layers can be rather small.
- The algorithm is restricted by the few solutions that have been found in the first layers. An MLGLS is a powerful algorithm that can easily find a larger number of

initial solutions to be used in subsequent layers, which can be beneficial if the retained reserve parameter is over 1230.

Both these problems could be solved if the population and the next layer gene pool are kept separate. This modification of an MLGLS is discussed in Chapter 5.

CHAPTER 5

Greedy multi-layered genetic local search with an expanding gene pool

5.0 Introduction

This chapter continues the investigation of the MLGA optimisation method suggested and implemented in Chapter 3. In Chapter 4 the MLGA was combined with a local search (LS) in order to improve its performance, with the resulting algorithm called an MLGLS. While the success rate of the combined algorithm was better than that of a basic MLGA by 10 to 20%, and the MLGLS proved to be up to five times faster; there are still more potential areas of improvement, which were identified in the end of Chapter 4. This chapter examines potential measures to enhance the search through the entire problem domain, without losing the clear structure and basic ideas of an MLGA.

5.1 Further improvement of the MLGLS algorithm

Three different MLGLS consisting of 7, 9 and 12 layers were trialed in Chapter 4, with the retained reserve parameter, R^0, ranging from 1220 to 1260 MW. As shown in Table 4.10, all three unit groupings performed well with R^0=1220 or 1230, with the success rate being over 90%. This allows us to propose that the MLGLS suggested and implemented in Chapter 4 is an acceptable algorithm for solving the case study problem with R^0<1240. Therefore, the experiments proposed in this chapter will pursue the development of an algorithm that can perform at a level beyond the MLGLS, with all trials to be conducted with $R^0 \geq 1240$, unless stated otherwise.

The MLGLS modifications examined in this chapter inherit positive characteristics of the basic MLGLS and MLGA algorithms, such as weeding out duplicates from the population, assessment of the candidates to the gene pool according to Criterion 3.2, etc. The GLS parameters used in this chapter have been selected in accordance with those used in Chapter 4.

5.1.1 *MLGLS with an expanding gene pool and a restricted elite*

While running the experiments described in the previous sections, it became clear that in some layers the MLGLS could have easily found many more sub-schedules suitable for the next layer gene pool than the currently required number of 300. On the other hand, if the population size is increased, it would lead to a longer running time in each generation and therefore the algorithm's total running time. A possible way to overcome this situation is to keep the gene pool separate from the population, and to expand the pool as needed. This keeps the population relatively small and at the same time enables the algorithm to find a large number of sub-schedules suitable for the next layer of an MLGLS.

Several issues should be considered while implementing an MLGLS with an expanding gene pool:

1. There still should be a limit on the size of the pool in case it grows too large. In the experiments described below it is taken to be 3000, unless stated otherwise.
2. The size of the elite part of the population should be restricted, to maintain diversity in the population. At the same time, it was empirically found that the next layer gene pool was filled faster when the elite part of the population was relatively large. For this reason, the following heuristic rules are implemented:
   a. The elite part of the population is initially 10% and is allowed to grow, as in the previous experiments, up to 50%.
   b. When the size of the elite is greater than 50% of the population, it is reduced down to 33%.
   c. New, randomly built individuals then replace the discarded elite.
3. To reduce the elite part, we need to determine which elite individuals are to go and which are to stay in the next generation. One solution is to use the *age* of an individual as a discriminator, that is, the number of generations the individual has been in the population. In this case, when the number of elite is reduced, the oldest individuals are the first to go. The notion of an individual's age was used, for instance, by Schwefel and Rudolph (1995). In their formal overview of evolution strategies, which can be applied to a GA, each individual has a special attribute,

being a *remaining lifespan,* which is reduced each generation. When the lifespan is zero, the individual is removed from the population. In the context of the restricted elite aspect of the algorithm, the age of an individual is used purely as the means of choosing part of the elite population to be discarded.

4. It becomes very important to ensure that a thorough exploration of the gene pool containing the sub-schedules found in previous layers is conducted, especially if the pool is much larger than the population. In some layers the MLGLS is rather slow in finding suitable candidates for the next layer gene pool and therefore, the heuristic rules 2.a-2.c could be waived because the elite part of the population is not large enough. Even though in theory every available gene value will find its way into the population in the fullness of time (Davis, 1991; Radcliffe, 1997), accelerating the introduction of new genes into the population is still necessary to speed up the search. Apart from increasing the mutation rate (Tate and Smith, 1993), this acceleration could be achieved by restricting the number of generations an individual is allowed to stay in the population, no matter what the size of the elite part of the population. This approach utilises the same idea of a limited lifespan suggested by Schwefel and Rudolph (1995). In this research the maximum lifespan is 5, unless stated otherwise.

Figure 5.1 presents the outline of an additional age management procedure that is used in the beginning of each generation of an MLGLS with an expanding gene pool in order to keep the population diverse and to explore the sub-schedules found in the previous layers.

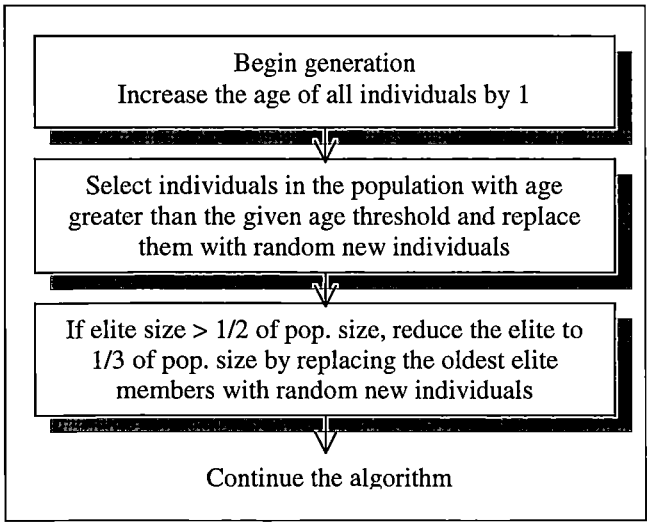

Figure 5.1 Age management procedure

The possible exception to this could be the first layer, where all individuals contain only single length genes and there is no need to explore an additional gene pool with long genes. Besides, in the first layer the search is relatively slow at the beginning, for example, according to Appendix 6, a 7-layer MLGLS finds the first sub-schedule in the first layer satisfying Criterion 3.2 with R^0=1240 in about 7^{th} generation. Therefore, the limited lifespan can cause a longer initial search for the first good individuals, which would slow down the entire layer. However, the restricted elite should be implemented in the first layer anyway, that is, if the elite part grows fast and reaches 50% of the population, it is reduced in accordance with the age management procedure illustrated in Figure 5.1.

5.1.2 Greedy MLGLS

When the neighbourhood of an individual in a population is searched, an LS may find several neighbours suitable for the next layer gene pool. However, the basic MLGLS described in the previous chapters selects only one of the best neighbours, discarding the rest. Since a GLS in each layer now aims to fill a large gene pool with as many sub-schedules as possible, it could be helpful if all solutions satisfying Criterion 3.2 were included into the pool, filling it faster and providing more potential genes for the next layer. This technique could be called a 'greedy GLS', as opposed to the basic GLS discussed earlier. When this greedy GLS is employed in an MLGA, the resulting MLGLS is also called 'greedy'.

It was confirmed by a number of experiments that the greedy approach is unsuitable for an MLGLS with the pool size equal to the population size. The greedy algorithm may quickly fill the population with too many neighbours of the same few individuals, reducing the chances of consequent layers finding satisfactory solutions.

It should also be noted that while all individuals satisfying pool Criterion 3.2 are placed in the next layer gene pool as a result of a greedy GLS, they are not placed in the population, except when replacing an individual that does not satisfy the pool criterion. This technique prevents a population with a relatively small size from being filled with a large number of similar individuals.

5.1.3 *Local search through the entire population*

In the basic MLGLS, an LS is performed only on individuals that do not satisfy Criterion 3.2 and, therefore, need improvement. Since a greedy MLGLS with an expanding pool tries to find as many good solution as possible, all individuals in the population should be examined for good neighbours. This can speed up the search by utilising all good sub-schedules found in the process and, consequently, reducing the number of LS operations in a run and decreasing the overall running time. For example, if an LS is performed on an elite member of the population already satisfying Criterion 3.2, the LS is more likely to find a large number of neighbours which can be included into the pool, than when searching through a neighbourhood of a poor quality schedule.

However, this elite individual also has other means of filling the pool with its descendants, by living and reproducing through several generations. Thus, if a large elite part of the population is extensively explored by an LS, the pool may be filled very fast with a large number of similar solutions, reducing the chances of success in the subsequent layers. This can be avoided if the number of neighbours sampled is reduced by half when a neighbourhood search is performed on elite individuals.

5.1.4 *Performance of the greedy MLGLS*

The following figures illustrate the effect of the new greedy GLS with age control on MLGLS performance. As an example, the 9-layer MLGLS that was suggested in Table 3.8 was tried, with the retained reserve parameter R^0=1240. Typical performance graphs from three of the layers were selected to illustrate some particular aspects of the algorithm.

The figures present two graphs each, the upper one showing the growth of the gene pool, while the lower graph presents the number of individuals in the population with a fitness of 1240 or more. It should be noted that since Criterion 3.2 is routinely used in Chapters 4 and this chapter, 'the number of individuals with fitness 1240' means 'the number of individuals with fitness 1240 and satisfying Criterion 3.2' throughout this

chapter. The individuals with the same fitness but not complying with Criterion 3.2 are of no account, unless stated otherwise.

Some additional information is presented in some figures, with Figures 5.2 and 5.3, for example, showing separately the number of individuals with fitness over 1240.

Figure 5.2 shows the fast growth of the gene pool in the first layer of the 9-layer greedy MLGLS.

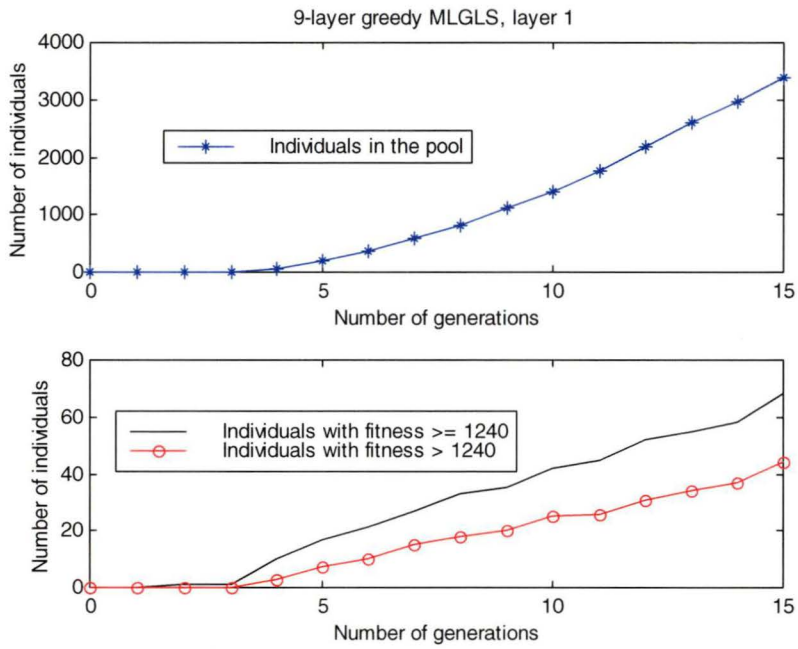

Figure 5.2 Gene pool growth in the first layer

The actual growth of good individuals in the population and solutions in the gene pool during the first five generations is presented in Table 5.1, since the scale in the upper graph is too large and cannot provide enough detail.

Table 5.1 Performance data for the first layer of a greedy MLGLS

Generation	Best fitness in the population	Number of individuals with fitness => 1240	Number of individuals with fitness > 1240	Number of sub-schedules in the gene pool
Initial population	1090	0	0	0
1	1160	0	0	0
2	1240	1	0	1
3	1240	1	0	12
4	1260	10	3	66
5	1260	17	7	198

As can be seen from the table, the first individual with fitness 1240 and satisfying Criterion 3.2 was found in the second generation and placed in the gene pool. During the third generation this individual remained the only one in the population with the fitness 1240. However, the GLS searched its neighbourhood and found eleven sub-schedules satisfying Criterion 3.2, which were also placed into the pool. In the fourth generation, nine more individuals satisfying Criterion 3.2 were found, three of them with a fitness greater than 1240. Additionally, during this search 66 sub-schedules were included in the pool. In the fifth generation the number of individuals in the population satisfying Criterion 3.2 grew to 17, while the pool included almost 200 sub-schedules.

As can be seen from Figure 5.2, the greedy GLS was able to find over 3,000 (exactly 3,395) sub-schedules for the next layer when the number of individuals answering Criterion 3.2 was just 68, with 44 of them having fitness greater than 1240. This shows once again the effectiveness of the LS and even more so, the effectiveness of the greedy version. Recall that in the first layer the age management procedure is not fully employed for the reasons discussed in Section 5.1.2.

Figure 5.3 shows the performance of the greedy MLGLS in the second layer and demonstrates the effect of the resticted pool. As well as the number of individuals with fitness equal to or greater than 1240, Figure 5.3 also presents the number of individuals deleted since the elite part of the population is too large.

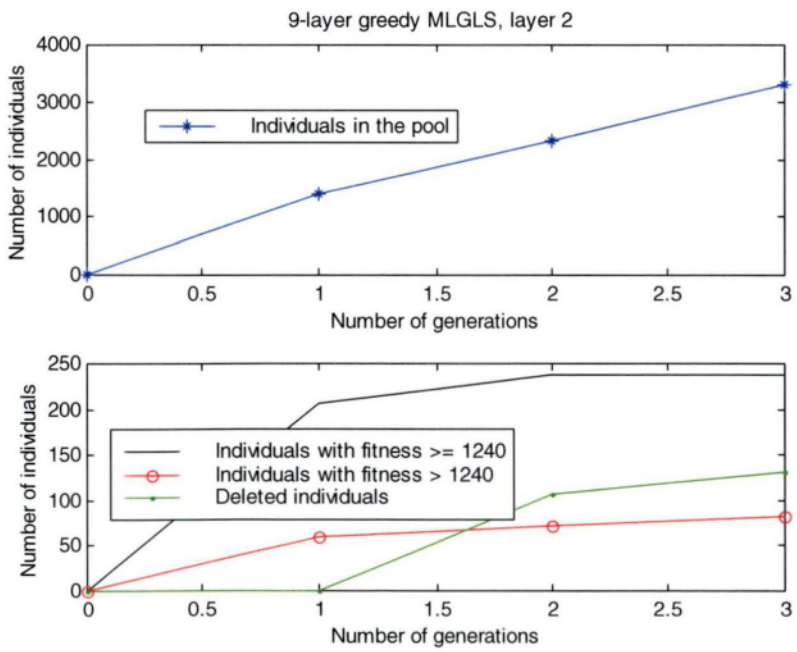

Figure 5.3 Gene pool growth in the second layer

As can be seen from Figure 5.3 it takes the algorithm only three generations to find over 3,000 solutions. Although there were no individuals with a fitness of 1240 or higher in the initial population, the greedy GLS improved over 200 individuals in the first generation while placing almost 1,500 sub-schedules in the pool. Since the elite part of the population was more than 50%, the age management procedure was employed in the second generation and over 100 elite members of the population were replaced by new random individuals. However, the elite grew again to almost 250 individuals while the pool increased to over 2,000 due to the effective search by the greedy GLS. In the last, third generation the picture is similar: despite about 130 elite members being replaced by random individuals, the elite part remained stable at a figure of about 240, while the pool included almost 1,000 new solutions.

There are other layers in an MLGLS that perform similarly to the second layer, that is, the algorithm concludes the search in a few generations and the elite size of the population is regulated by replacing part of it by random individuals. These layers include the last two in the 9-layer MLGLS and sometimes the third layer. Obviously, the greedy search could have filled the gene pool even faster without the regular discarding of part of the elite. However, such tactics would lead to a smaller number of long genes from the layer's gene pool to be processed, which may, in turn, result in less variety in the individuals in subsequent layers. While the aim of a multi-layered algorithm is to direct the search into some promising regions of the vast problem domain, it should not narrow the search efforts to just a few areas around the solutions found in the previous layer. The experiments showed that a significant variety in the gene pool improves the algorithm's chances to succeed. This is why the greedy MLGLS employs the age management procedure even when the GLS is quite efficient without it.

Figure 5.4 demonstrates the greedy GLS behaviour in the fourth layer, which is slightly different from the previous layers. As shown in the figure, the search in the fourth layer is slower and takes 54 generations to find 3,000 solutions. During the first five generations the number of individuals with fitness 1240 does not reach 50% and therefore the age control procedure is employed for the first time in the sixth generation to replace 179 individuals with an age greater than the age threshold (five generations) by new individuals, randomly built from the sub-schedules found in the previous layer. As shown in the figure, the number of individuals deleted in the sixth generation is greater than the number of elite members in the population.

The next peak in the number of deleted individuals occurs in the twelfth generation, when it was found that 92 individuals had been in the population for more than five generations. In later generations the age control procedure results in the deletion of 14 to 67 individuals every generation, which keeps the number of solutions with a fitness of 1240 between 40 and 80. During the run, the size of the pool increases at fairly constant rate.

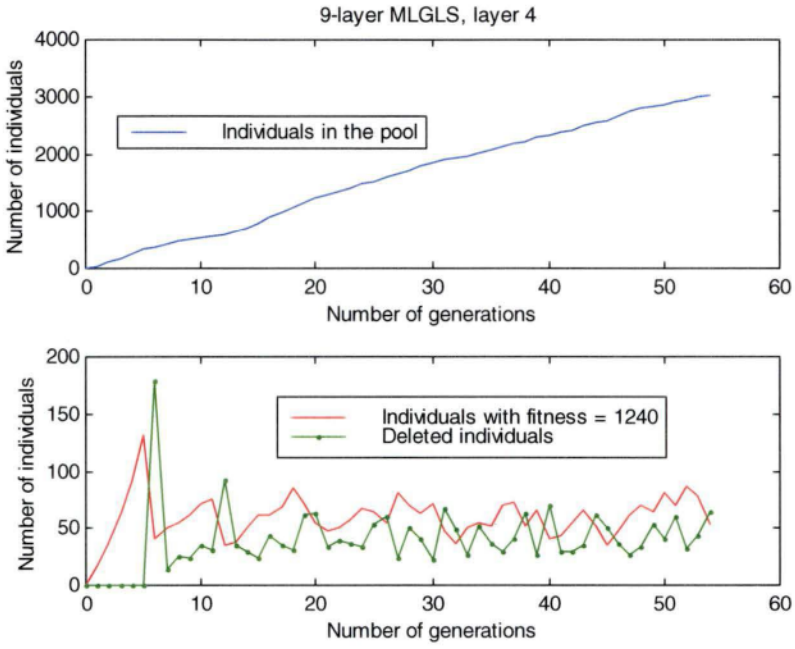

Figure 5.4 Gene pool growth in the fourth layer

Figure 5.4 presents an example of the long gene pool exploration generated by regularly introducing new individuals into the population based on their age. Sometimes a mixture of the age control measures is displayed, when at some stages the elite part of the population grows fast and is then reduced if exceeds 50%, while at other times the elite part is small and new individuals replace the ones with an expired lifespan.

5.2 Implementation of the greedy MLGLS

5.2.1 Maintaining the age diversity in the population

In the algorithms that were discussed in Chapters 3 and 4, the crossover rate was 1 and as a result, the entire non-elite part of the population participated in reproduction,

providing a fast and thorough exploration of the layer search space. If the same approach is implemented in an MLGLS with an expanding gene pool, the entire non-elite part of the population will always be age 1. If in some layer the search is relatively slow and the number of individuals satisfying Criterion 3.2 is small, the age management procedure will change only a few elite members in each generation.

For the age management procedure to perform an effective exploration of the layer search space, individuals in the population should be of various ages, with the number of new individuals in the population kept relatively low. This can be done, for example, by reducing the crossover rate to 0.5 – 0.7. This way a part of the population will stay unchanged every generation and as a result, individuals of various ages will always be present.

Alternatively, the genetic operators can be modified so that the individuals produced by these operators inherit the remaining lifespan of a parent together with its long gene. In this case a new individual will have the maximum lifespan only if the mutation operator has replaced its long gene. Such modification of the operators makes an individual's age equal to the number of generations its long gene has been in the population. Of course, when new individuals are randomly built from the available genes, some long genes can again enter the population and receive the maximum lifespan.

5.2.2 *Weeding interchangeable sub-schedules from the gene pool*

We may recall that in an MLGA and a basic MLGLS, interchangeable sub-schedules are allowed to stay in the population until the end of a layer run, when the entire elite part of the population is transferred into the next layer gene pool (see Section 3.3.2). Since a gene pool in either algorithm, including a greedy MLGLS, should contain only individuals that are not interchangeable, a regular elimination of interchangeable sub-schedules should be conducted. Note that by doing this, the duplicates are discarded too since they present a special case of interchangeable sub-schedules.

On the other hand, if the pool is very large, this procedure would take a considerable amount of time. For example, in some of the experiments performed in this research up to 5,000 candidates for the gene pool were found in one generation, and then they had to

be checked against a gene pool already containing almost 3,000 sub-schedules. Many of these candidates could be identical, and measures were taken to reduce their number, by, for example, reducing the number of neighbours sampled when performing the LS. However, a faster procedure for finding interchangeable sub-schedules is necessary even when only a hundred new sub-schedules are added to a gene pool of almost 3,000.

Several algorithms for weeding out interchangeable sub-schedules were trialed, and the fastest one was incorporated along with the rest of the research software, on a standard desktop computer using the MATLAB programming language.

The algorithm consists of two separate steps.

- First, the sub-schedules that are candidates to the pool are rearranged. This is done by selecting a group of genes corresponding to identical units (that is, having the same capacities and requiring the same number of weeks for maintenance) and rearranging the values of these genes in ascending or descending order. After the rearrangement is performed in all groups of identical units, the interchangeable sub-schedules become duplicates.
- The second part of the algorithm is the actual weeding of duplicates from the set of all candidates to the pool. If a candidate individual differs from the sub-schedules in the pool it is added to the pool. At the same time the individual is compared to the other candidates and all identical candidates are discarded, thus reducing the number of sub-schedules to be compared to the ones in the pool. This algorithm is shown in Figure 5.5.

input: *Pool, NewPool*
select *S* **from** *NewPool*
if $Pool = \varnothing$, **move** *S* from *NewPool* to *Pool*
else
    $P = Pool \cup NewPool$
    for all genes *g* in *S* **or until** $P = \varnothing$
        discard from *P* inds. with the corresponding gene $\neq g$
    end % for
    if $P \cap Pool = \varnothing$, **move** *S* from *NewPool* to *Pool*, **end**
    NewPool = *NewPool* - *P*
end % if
repeat until $NewPool = \varnothing$

Figure 5.5 Algorithm for weeding duplicates

5.2.3 *Termination criteria*

As in the other algorithms, the foremost termination criterion is finding a specified number of good solutions. The maximal number of generations was set to 100 and as before, three termination variables have to be determined for evaluating the algorithm's performance, $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$. As was explained in Section 3.3.4, the GLS performance is evaluated every $\mathcal{T}_2$ generations starting at $\mathcal{T}_1$ generations. The algorithm is stopped if the increase in the number of individuals suitable for the next layer gene pool during the last $\mathcal{T}_3$ generation is less than 10%. If the gene pool is empty, the run is declared unsuccessful. In Chapter 4, for example, the termination variables were assigned values of 5, 20 and 5 respectively.

As was discussed in Section 3.4.1, the most important variable for the algorithm's performance is $\mathcal{T}_1$. If it is too small, the search space of the layer will not be examined thoroughly, while a large value of $\mathcal{T}_1$ may lead to an unnecessarily long search in a space that simply does not have a large number of good solutions. Now, when the use of an LS operator and the restricted lifespan of the individuals ensure an efficient search, $\mathcal{T}_1$ can be adjusted according to the size of the gene pool available to the algorithm. Clearly, if a gene pool contains 3,000 sub-schedules, $\mathcal{T}_1$ should be larger than when the previous layer has found only ten sub-schedules. In the experiments conducted here the following rule was used to determine $\mathcal{T}_1$ in a layer $k>1$:

$$\mathcal{T}_1^{(k)} = c \times \left(\text{round}\left(\left| \mathcal{S}^{(k-1)} \right| / S_P \right) + 1 \right) \times \theta , \tag{5.1}$$

where c is a constant coefficient, $\left|\mathcal{S}^{(k-1)}\right|$ is the number of sub-schedules in the layer's gene pool, S_P is the population size, and θ is the maximum lifespan for the individuals in the population. Both θ and c can vary in different layers, as long as the value of $\mathcal{T}_1$ provides adequate exploration of the layer's search space and complies with the principle of diminishing returns if the algorithm runs for a long time. In the experiments conducted in this research θ=5 and c=1 unless stated otherwise. At the same time $\mathcal{T}_2 = \mathcal{T}_3 = 5$.

The following figures illustrate the effect of the new termination criteria on the search performed by the greedy MLGLS. As examples, two layers of a 9-layer algorithm and one layer of a 7-layer algorithm are taken to demonstrate some specific aspect of the new termination criteria. Note that the examples presented in figures 5.2-5.4 do not depend on the criteria and would have been exactly the same even if $\mathcal{T}_1$ was equal to 20, as in Chapter 4. This is because in all three examples the number of sub-schedules in the pool was growing steadily at a rate of more than 10% every five generations until it reached 3,000. Besides, in the first two cases the pool was filled in less than 20 generations. The termination variables described above have more evident effect in a case when the search is unproductive and the gene pool for the next layer is filled slowly.

Figures 5.6 shows the performance of the seventh layer of the 9-layer greedy MLGLS. The algorithm in the previous layer found only 465 sub-schedules for the 9th layer gene pool and, according to Eq. (5.1), $\mathcal{T}_1$=15.

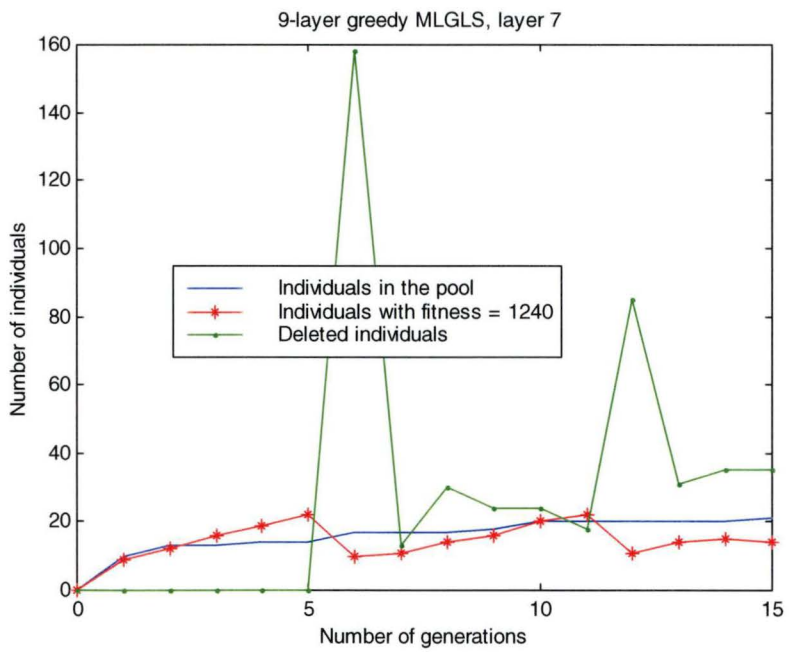

Figure 5.6 The greedy MLGLS performance with $\mathcal{T}_1$=15

As shown in the figure, the pool size grew fast in the first two generations (10 and 13 solutions in the pool), after that the growth slowed down. Note that in the third generation the number of individuals with fitness 1240 in the population is larger than the pool size, because interchangeable individuals stay in the population without being included in the pool. Just one more good sub-schedule was included in the pool in

generations 3 to 5, however, as shown in Figure 5.6, a massive influx of new individuals into the population occurred in the sixth generation as a result of age management procedure. It allowed the algorithm to find three more good solutions right away and another three before the tenth generation. Since $\mathcal{T}_1$=15, in the fifteenth generation the pool size growth was reviewed for the last five generations and the algorithm was stopped because during that time only one new sub-schedule was added to the pool.

Next in Figure 5.7, a different situation is presented. There, the third layer of the 7-layer MLGLS received the gene pool from the previous layer with 3,042 long genes in it. This resulted in $\mathcal{T}_1$=55, according to Eq. (5.1).

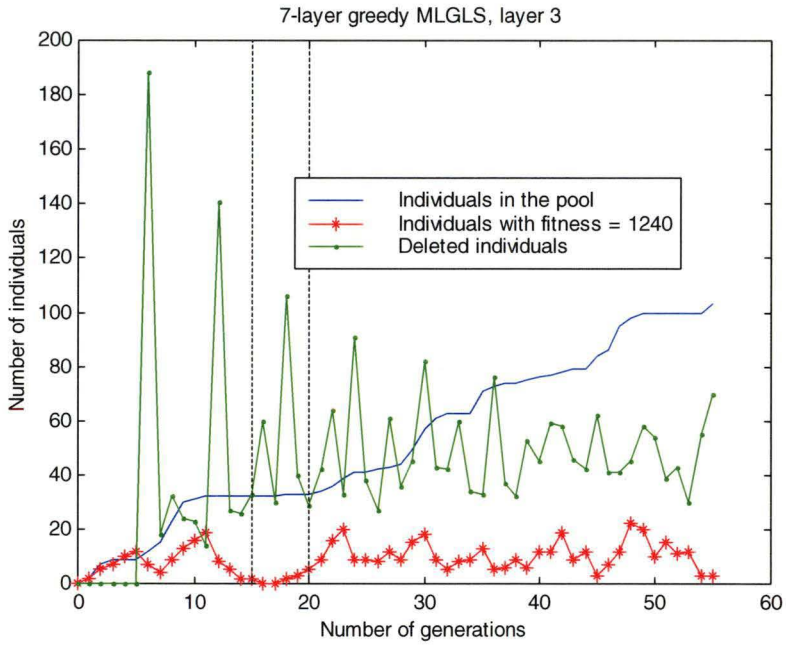

Figure 5.7 The greedy MLGLS performance with $\mathcal{T}_1$=55

As can be seen from the figure, the number of sub-schedules in the pool grows very slowly. For example, if $\mathcal{T}_1$ was 20 generations, the algorithm would have stopped there because between the 15th and 20th generations only one new sub-schedule was found. However, as seen from the figure, generations 10 to 20 were the least successful of the entire run, with the slowest pool growth. Later along the run, there were several boosts in the pool size. For example, in generations 28 - 35 the algorithm was able to find up to eight sub-schedules per generation. The last large addition was in the 47th generation, when nine new sub-schedules were added to the pool. The search continued for 55 generations, as required by $\mathcal{T}_1$=55, and while in the last generation three more solutions

were found, the growth rate was less than 10% in the last 5 generations and thus the layer was considered completed.

It is possible that the algorithm would have found some more solutions if allowed to run even longer; however, the 55 generations is probably enough to sufficiently explore the layer's long gene pool. Alternatively, if it is known from previous experiments that the search in a certain layer is usually slow, the coefficient c in Eq. (5.1) can be increased for this particular layer.

Obviously, when $\mathcal{T}_1$ is determined according to Eq. (5.1), it can be rather large if the gene pool filled by the previous layer is large. In order to reduce the number of generations in a layer, $\mathcal{T}_1$ can be modified after the number of solutions found becomes larger than a given threshold. For example, if the number of solutions found by the greedy MLGLS is more than half of the maximum pool size while the generation number is less than $\mathcal{T}_1$, then $\mathcal{T}_1$ is reduced to the closest multiple of five. Figure 5.8 illustrates the result of such an adjustment.

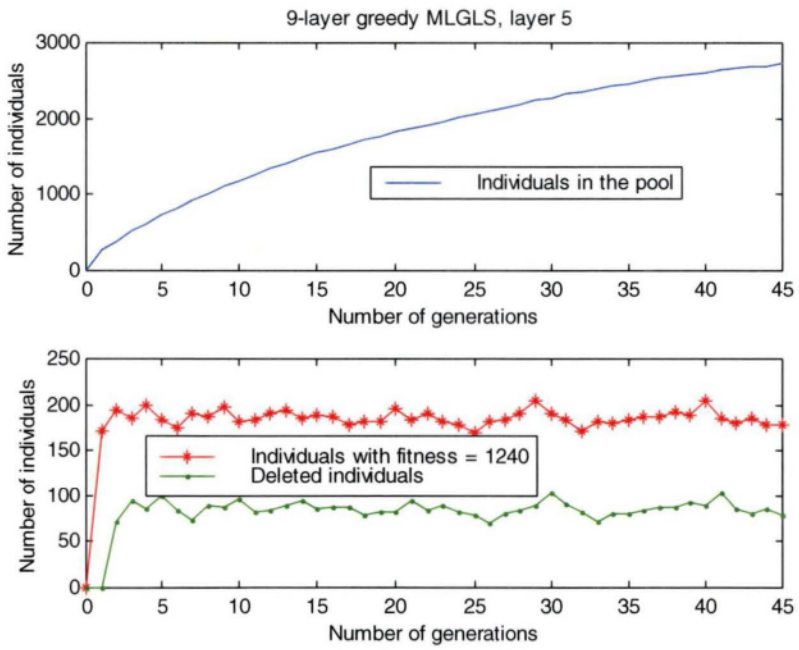

Figure 5.8 $\mathcal{T}_1$ adjustment

The figure represents the performance of the fifth layer in the 9-layer greedy MLGLS. In this particular run the previous layer has found over 3,000 sub-schedules for the fifth layer gene pool and according to Eq. (5.1) $\mathcal{T}_1$=55.

As can be seen from the figure, there are about 200 good individuals in the population in each generation, and as a result, up to 100 individuals are replaced by new ones. At first, the number of good solutions found by the fifth layer grows fast, but gradually the growth rate slows down, the reason being that the long genes in the layer's gene pool have been explored thoroughly already and completely new sub-schedules are found less frequently. However, in the 15^{th} generation the number of good sub-schedules reaches 1550 and $\mathcal{T}_1$ is reduced from 55 to 20, which means that the layer's performance is being evaluated every five generations starting at the 20^{th} generation. After 45 generations, when the search becomes too slow, the algorithm is stopped. This allowed the algorithm to avoid running for another ten unproductive generations while the next layer still received a large number of sub-schedules in its gene pool.

5.3 Greedy MLGLS preliminary results

The greedy algorithm proves to be an efficient way of exploring the problem search space. There are three main points of interest when the experimental results of the greedy MLGLS are discussed.

1. Improvement in the performance with R^0=1240

The experimental results of the greedy MLGLS with the retained reserve parameter 1240 are presented in Appendix 7. Table A7.1 contains the results of the 7-layer greedy MLGLS while Tables A7.3 and A7.5 present the results of the 9 and 12-layer algorithms. As shown in the tables the algorithm performance has significantly improved when compared with the basic MLGLS discussed in Chapter 4. For example, the greedy MLGLS with 7, 9 and 12 layers provides a success rate of 85%, 77.5% and 40% respectively, whereas the basic MLGLS has a success rate of no more than 30% (see Table 4.10).

If the data presented in Appendix 7 is examined, it is clear that the LS component of the algorithm was used to its full potential and in all layers the greedy MLGLS was able to find a large number of solutions, consistently more than the population size of 300. This is especially true in the initial layers and the very last layers. For instance, the 12-layer MLGLS produced over 8,000 different solutions in just one generation in

successful runs. Note that the number of converged genes is reduced from over 20 in the basic MLGLS to about 12 on average, which shows a larger variety in the resulting solutions.

2. *Greedy MLGLS results with various* R^0

The greedy algorithm was also tried with retained reserve values greater than 1240 MW and the summary of the experimental results is given in Table 5.2. As a result of the greedy MLGLS efficient search, the algorithm was able to attain results with a retained reserve parameter of 1250, even though this only happened once or twice during 40 runs of each unit grouping, which was never achieved with the basic MLGLS. This shows that the greedy variant of the algorithm is more effective even with harder constraints on the problem.

Table 5.2 Greedy MLGLS performance

Retained reserve, R^0	12-layer MLGLS		9-layer MLGLS		7-layer MLGLS	
	Successful runs out of 40	Number of generations in a run	Successful runs out of 40	Number of generations in a run	Successful runs out of 40	Number of generations in a run
1240	16 (40%)	180.70	31 (77.5%)	183.88	34 (85%)	183.28
1250	2 (5%)	134.96	1 (2.5%)	137.40	2 (5%)	152.35
1260	0	118.45	0	119.33	0	137.08

It can be noticed that the algorithm performance with R^0=1240 is slower than with R^0>1240, according to the table. However, the number of generations in a run was taken as the average of all runs and when R^0>1240 most of the runs were unsuccessful and did not run through all the layers. In the successful runs with R^0=1250 the number of generations was 170-220.

3. *Efficiency due to an extensive search*

The greedy algorithm is more efficient and capable of finding a large number of solutions, but the new algorithm is relatively slow when compared to the basic MLGLS. For example, the greedy MLGLS took about 180 generations on average while the basic

algorithm needed only about 113 generations, according to Table 4.10. Besides, each generation in the greedy algorithm takes slightly longer than an average generation in the basic one due the following reasons:

- The elite part of the population is kept small which requires the evaluation of new additional individuals in each generation, unlike in the basic MLGLS where the elite part grows up to the population size.
- The greedy MLGLS performs the search in the neighbourhoods of all individuals rather than only non-elite individuals.
- Discarding interchangeable solutions from the gene pool can be time-consuming if the gene pool is large.

Therefore, it can be stated that the significant increase in efficiency of the algorithm is achieved by a more thorough exploration of the problem search space which takes almost twice the time.

At the same time it is clear that the processing time of the 9 and 12 layer MLGLS can be decreased, if the number of required solutions was reduced to, say, 1,000 in the first layers. This is because the subsequent layers are very fast and efficient and find up to 3,000 solutions in each generation, thus leaving a large part of their gene pools unexplored.

The greedy algorithm proves to be an efficient way of a problem search space exploration. However, due to the large number of sub-schedules it tries to find in each layer, some layers are rather slow. Therefore it would be advantageous if it was possible to reduce the search time while retaining this efficiency.

5.4 Population initialisation with a schedule builder

5.4.1 Combining two types of representation in one algorithm

The greedy MLGLS needs a large number of generations to explore the layer's search space and find a required number of solutions. At the same time there is a well-known way to speed up the search by seeding the population with good solutions found in previous runs of the algorithm or by some other methods (Greffenstette, 1987;

Eshelman, 1997). In fact, Chapter 2 of this research described experiments with a traditional GA seeded by solutions found by a schedule builder. It was found that the seeded GA was able to fill the population with individuals with the same objective value but failed to improve that value due to the inability of a traditional GA to examine the large search space of a problem (see Section 2.4.3).

Alternatively, some problem-specific heuristics can be used to initialise the population (Ibaraki, 1997). For example, it could be possible that by initiating the greedy MLGLS population with a good heuristic the algorithm's starting conditions may improve which in turn will upgrade the algorithm's performance.

In Chapter 2 the direct and indirect problem representations were examined. The direct representation was chosen for the MLGA and MLGLS algorithms because it describes the entire domain of a problem. However, the direct representation unnecessarily expands the algorithm search space by also including individuals of poor quality.

On the other hand, as was explained in Chapter 2, an indirect representation with a schedule builder has the following characteristics:

- The high quality of the schedules when compared with a direct representation.
- The inability to represent the entire problem domain.
- High redundancy of the indirect representation, that is, a schedule builder can convert two different sequences of units into the same schedule.

The advantages of the two representations can be combined and used to the benefit of the resulting algorithm, if a schedule builder from the indirect representation is used as the means to initialise a directly represented population. A similar approach, for example, was used by Bruns (1993) to solve a JSS problem, when individuals directly representing production plans were initialised with the help of problem-specific heuristic rules that were, essentially, a variant of a schedule builder. Such initialisation ensured the feasibility of all schedules in the population.

If this approach is used in the case study, the greedy MLGLS needs only one minor adjustment when new schedules-individuals are created, that is, either at the initialisation phase or when some individuals in the population are replaced by randomly built new ones, for example, during weeding out the duplicates or individuals with the expired lifespan. To create new individuals, random sequences of units are

presented to a schedule builder which transforms the sequences into schedules. The resulting schedules will always be legal if the 'soft' schedule builders are used since they produce individuals that comply with the problem's constraints (2.6), as was shown in Section 2.4.2.

If a layer $k>1$ is considered, a schedule builder is given an arbitrary 'long' gene from the available gene pool as well as a random sequence of units that are to be scheduled in this layer. The nett reserve provided by the long gene is taken into account by the schedule builder when placing the units from layer k for maintenance.

After the initialisation the newly built individuals are treated as ordinary directly represented schedules and undergo one or two-point crossover and poolwise mutation. The resulting individuals will be built from legal genes but unfortunately, not necessarily represent legal schedules as was explained in Section 2.4.3. However, it is important to employ the ability of the direct representation operators to provide the genetic search through the entire layer domain and not only among the schedules that a schedule builder can produce. The LS operator is employed to search the neighbourhoods of the individuals as was described in Chapter 4, and if the LS is used after the genetic operators, it can partially repair the damage and even improve the individual further.

In the experiments conducted in this research both schedule builders, 'first available' and 'deepest first' were used for initialisation at the same time. However, since the 'first available' schedule builder proved to be more successful (see Chapter 2), it was used more often, with a probability of 0.8, while the 'deepest first' schedule builder was employed with a probability of 0.2.

5.4.2 The effect of the combined representation on greedy MLGLS performance

The layered experimental results with schedule builder initialisation of individuals are presented in Appendix 8. The experiments were conducted with the three unit groupings given in Tables 3.3, 3.8 and 4.3. Appendix 8 presents the results with the retained reserve parameter $R^0=1240$ and 1250. The structure of the tables in the appendix is the

same as in the previously discussed Appendices, including Appendix 7. As before, the data in the tables is averaged over 40 runs for each experiment.

When the data presented in Appendixes 7 and 8 is analysed, there are several differences in the performance of the greedy MLGLS with direct initialisation versus indirect initialisation (with the schedule builders):

- The employment of the schedule builders for initialisation resulted in a significant increase in the fitness of the initial population. For example, if the 7-layer algorithms are considered with R^0=1240, the employment of the schedule builders for the initialisation of individuals increased the best fitness in the first layer from about 1050 to 1260 according to Tables A7.1 and A8.1. At the same time the average fitness in the initial population grew from 511 to 1209.
- Note, however, that with the schedule builder initialisation the subsequent populations may contain individuals with lesser fitness because the genetic operators are rather disruptive and do not guarantee the legality of the resulting schedules (see Section 2.4.3). This situation is reflected in Table A8.1 when in the first layer the average fitness at the end of the run, about 1125, is less than in the initial population. A similar situation exists in the other layers of the 7-layer algorithms as well as in the 9 and 12-layer ones with various values of the retained reserve parameter.
- A large number of solutions were found in each layer. For example, if the performance with R^0=1240 is considered, the algorithm with the schedule builders was able to find more solutions satisfying Criterion 3.2 than the algorithm with direct initialisation in the corresponding layers, according to Tables A8.1, A8.3 and A8.5.
- The search by the modified greedy MLGLS was quite productive when running the experiments with R^0=1250. As shown in Table A8.4, the 9-layer greedy MLGLS achieved 50% success in 40 runs, with the 7-layer algorithm having a similar success rate of 47.5%, according to Table A8.2. These are the best results that were obtained by any algorithm examined in this research.
- As a result of the enhanced initialisation, the algorithm with the schedule builders becomes faster in most layers, especially with R^0=1240.

Performance of the 9-layer greedy MLGLS with R^0=1240

In order to examine the effect of the indirect initialisation in detail, a comparison of two greedy 9-layer MLGLS algorithms with R^0=1240 is presented in Table 5.3. The table includes some extracts from the Appendix 7 and 8 tables plus some additional data such as the number of individuals satisfying Criterion 3.2 ('good' individuals) in the initial population with combined representation. Also included is the average number of individuals removed in each generation of a layer by the age management procedure (see Section 5.1.2).

Table 5.3 Comparison of the two greedy 9-layer algorithms with R^0=1240

Layer	Greedy MLGLS with direct initialisation				Greedy MLGLS with indirect initialisation			
	Number of generations in a run	Generation of the first 'good' individual	Number of individuals removed	Pool size at the end of a run	Number of generations in a run	Number of 'good' individuals in the initial generation	Number of individuals removed	Pool size at the end of a run
1	15.94	2.03	0.00	3163.90	8.18	248.41	141.68	3162.95
2	4 00	0.65	73.45	3299.85	1.00	281.91	91.23	3481.45
3	9.06	0.63	79 45	3127.08	2.00	267 64	119.23	3788.36
4	53 03	1.00	49 25	2133.25	4.36	87.18	47.50	3451.14
5	36 74	0.93	64 45	1772.63	7 09	204 86	148.09	3128.36
6	41 58	1.00	47 35	807.33	30.14	52.68	59 91	2939.95
7	28.52	0.74	34.53	526.16	48.59	38.73	44 18	1883.73
8	5 90	0.84	86.91	2740.28	1.00	279.18	89.73	3862.05
9	2.84	1.00	50.09	3953.32	1.00	253.64	77.14	3927.14

After examining the table the following features of the algorithm with the schedule builder initialisation can be noticed:

- **Large proportion of good individuals in the initial population.** The employment of the schedule builders resulted in a large proportion of the initial population consisting of individuals that satisfy Criterion 3.2. For example, in the second layer this proportion averaged over 90%. By contrast, when direct initialisation was used, the first good individual was usually found only after an LS operator had improved the first generation, with only an occasional good individual in the initial population.

- **Faster and more efficient search in most layers**. The better starting conditions of the algorithm with the schedule builders were further emphasised when an LS operator examined the neighbourhoods of all good individuals, finding a large number of solutions suitable for the next layer gene pool and filling the pool in just a few generations in most layers. For instance, in the fourth layer a larger number of solutions (over 3,400) were found in about four generations instead of just over 2,000 solutions found in over 50 generations without schedule builder initialisation.

- **More thorough search in slow layers**. The only layer where the algorithm with the direct initialisation took fewer generations than the algorithm with the schedule builders was layer No.7, with the average number of generations being about 30 compared to almost 50. This layer was the most difficult for the 9-layer greedy MLGLS, as in the experiments with the direct initialisation all previous layers were successful in all 40 runs, however 8 runs of the 7^{th} layer failed, as shown in Table A7.3.

  When the schedule builders are used, they produce on average about 40 good individuals in the initial population, which is the lowest number in all layers, according to Table 5.3. Clearly, the concentration of good individuals in the population is rather low, and in order to be successful the algorithm has to perform a thorough search through the whole search space, including the gene pool containing sub-schedules found in the previous layers. This process can take a long time if the gene pool is large.

  When the schedule builders have been used for initialisation, the gene pool received by the 7^{th} layer averages almost 3,000, compared with only about 800 in the algorithm with the direct representation. Because the termination criteria are modified depending on the pool size (see Section 5.2.3), the more efficient algorithm runs 60% longer but with 100% success.

- **Faster rotation of the long genes in the population**. Another aspect of the algorithm that is presented in Table 5.3 is the number of individuals deleted from the population as a result of the age management procedure. As can be seen in the table, the algorithm with the schedule builders is more active in searching among the sub-schedules provided by the previous layers, deleting on average more individuals than the algorithm with direct initialisation. In most layers of the 9-layer algorithm the large number of replaced individuals was due to the efficiency of the

schedule builders, since they produced too many good schedules which had to be replaced regularly. However, this tendency of a fast long gene rotation is also present in all experiments with R^0=1250 where the proportion of good individuals in the population is lower. In this situation the fast rotation resulted in a better success rate, as is shown below.

Performance with retained reserve parameter of 1240 and 1250

Table 5.4 summarises the effect of the indirect representation in all experiments with R^0=1240 and 1250. It shows the success rate and the number of generations in a run for all three types of layering arrangements.

Table 5.4 Greedy MLGLS comparison

Number of layers in the MLGLS	Retained reserve, R^0	Greedy MLGLS with direct initialisation		Greedy MLGLS with indirect initialisation	
		Successful runs out of 40	Number of generations in a run	Successful runs out of 40	Number of generations in a run
7	1240	34 (85%)	183.28	40 (100%)	104.85
	1250	2 (5%)	152.35	18 (45%)	152.45
9	1240	31 (77.5%)	183.88	40 (100%)	103.36
	1250	2 (5%)	137.40	20 (50%)	182.13
12	1240	16 (40%)	180.70	40 (100%)	130.73
	1250	2 (5%)	134.96	4 (10%)	145.28

As shown in the table, the modified greedy MLGLS had a 100% success rate with R^0=1240, no matter whether the MLGLS consisted of 7, 9 or 12 layers. This is a significant improvement when compared with the results of all other algorithms discussed in this thesis. Obviously, the employment of the schedule builders for creating new individuals increased the greedy algorithm's efficiency, boosting the success rate by 2.5 times in the case of the 12-layer greedy MLGLS with R^0=1240. In the other two types of layering cases the schedule builders simply provided a better initial search, thus increasing the success rate to 100% with the same R^0. Moreover, the number of generations in a run was reduced by about 40% in both the 7 and 9-layer algorithm; and by almost 30% in the 12-layer algorithm.

As shown in Table 5.4 and, in more detail, in Table A8.4, the 9-layer greedy MLGLS achieved 50% success in 40 runs, which was previously unattainable. The 7-layer algorithm had a similar success rate of 47.5%, as shown in Tables 5.4 and A8.2. These are the best results that were obtained by any algorithm examined in this research.

The results of the 12-layer greedy MLGLS with R^0=1250 are not so impressive, with only 10% of the runs successful. This can be explained by the fact that with the smaller number of units scheduled in each layer, Criterion 3.2 cannot control the quality of the solutions as effectively as with the 7 and 9-layer greedy MLGLS, which have a larger number of units in each layer.

Performance with retained reserve parameter of 1260

Unfortunately, neither algorithm was able to find any solutions with R^0=1260. When the 9-layer greedy MLGLS with the schedule builder initialisation was tried, the first three layers were able to schedule 22 units quite easily in under 10 generations per layer, finding the required number of solutions. However, the next, fourth layer, running for 60-70 generations and adding four new units managed to find on average only about 100 solutions (the absolute maximum number was 497). Then the fifth layer, which scheduled only two additional units, when successful, was able to build no more than 328 sub-schedules, with the average number being only 50. The sixth layer with the next three new units was never successful.

Similar results occurred with the 7-layer algorithm. 24 units from the two first layers were scheduled easily, however the third layer with four units was at times unsuccessful and at the very most was only able to find 340 solutions with the average number being 20. Layer 4 was always unsuccessful.

The experiments with R^0=1260 were run with a variety of parameters, including GLS parameters (see Chapter 4), genetic operators probabilities, maximum lifespan equal to 10 generations and the coefficient $c=1.5$ to 2 for Eq.(5.1), which resulted in additional running time for the algorithm (see Section 5.2.3). A number of different unit layering patterns were used with similar results.

These results indicate that there are no solutions that provide the nett reserve of 1260 MW at all, or if they exist there are only a small number of them, which the MLGLS

has been unable to locate. It is still possible that through some advanced initialisation heuristics or search technique some variant of GAs would be more successful, but it is believed this research has identified most significant improvements to GAs at this time. A further line of research could be to use the algorithm suggested and developed in this research as the first stage in an optimisation process, and then use some other technique to select the best suitable solution and, perhaps, improve it further.

Possible fine-tuning of the algorithm:

- **Smaller gene pools in some layers, especially with 1240.** From Table 5.3 it can be observed that the algorithm's speed can be increased even further without reducing its success rate, if the required gene pool size for the next layer is decreased in some layers to 1,000 or even down to 500. For example, as shown in the table, the second layer was completed in just one generation, which means that the gene pool with long genes was sampled only twice: when initialising the population and in the first generation. Therefore, it is unnecessary to search for 3,000 solutions in the first layer. Similarly, the required pool size can be reduced in other layers that pass this pool to fast layers with easily scheduled units. However, when $R^0=1250$, the search becomes slower and such a reduction in the pool size may result in a less successful performance.
- **Different lifespan.** As explained earlier, the maximum lifespan in the population was taken to be 5 generations. A number of experiments were conducted with the lifespan equal to 10 to find out whether a longer lifespan would be beneficial, especially with $R^0=1250$. These experiments did not show any difference in the success rate of the algorithm, however, the number of generations in a run did increase. Conversely, reducing the lifespan to 2 or 3 generations reduced the success rate, especially with the 7-layer algorithm, and in all algorithms with $R^0=1250$.
- **Different layering of the units.** There are other possible ways of layering the units. For example, experiments were conducted with a 6-layer algorithm, in which the first five layers were exactly the same as in the 7-layer algorithm, with the last two layers then combined into one. This layering has a similar success rate, but took slightly longer to finish the last layer compared to the 7-layer algorithm's last two.

5.5 Conclusion

This chapter examined a further improvement of an MLGLS, the greedy MLGLS with the expanding gene pool, which increased the overall efficiency of the algorithm. By using a schedule builder for initialisation the algorithm performance was even further enhanced, and the number of generations in a run was reduced.

Table 5.5 compares the performance of a single-layer GA with indirect representation, an MLGA, a basic MLGLS and a greedy MLGLS. Data in the table presents the best performance of each of these algorithms.

Table 5.5 Comparison of the best results of different algorithms

Retained reserve parameter, R^0	1-layer GA with indirect representation		9-layer MLGA		7-layer MLGLS		9-layer greedy MLGLS with indirect initialisation	
	Success rate	Number of generations in a run	Success rate	Number of generations in a run	Success rate	Number of generations in a run	Success rate	Number of generations in a run
1220	100%	64.41	100%	442.35	100%	62.10	100%	11.75
1230	100%	141.93	28%	675.38	97.50%	83 70	100%	12.88
1240	92 50%	200.20	8%	720.80	27.50%	113.25	100%	103.36
1250	0	n/a	0	n/a	0	n/a	50%	182.13
1260	0	n/a	0	n/a	0	n/a	0	n/a

As can be seen from the table, the greedy MLGLS with indirect initialisation is the fastest and most efficient. When compared, for example, with the single-layer GA with indirect representation, the greedy algorithm is not only more successful with the retained reserve parameter, R^0, of 1240 and 1250, but it is twice faster with R^0=1240 and five to ten times faster with R^0 less than 1240. Note also, that when the greedy MLGLS is successful, it produces over 3,000 solutions, which is ten times more all other algorithms. However, the most important aspect of its performance is that it was able to produce schedules with a nett reserve of 1250 MW with a 50% success rate, while no other algorithm could produce even a single schedule at this level.

The greedy MLGLS algorithm has the following advantages:

- The problem is divided into layers and, therefore, the search is directed into several promising areas instead of a purely random search through the vast problem domain.

- These promising areas vary in every run, thus the search is conducted in the entire problem domain and not in the area determined by the heuristic schedule builders as in the indirect representation.
- The schedule builder initialisation speeds up the search by providing directly represented individuals of good quality. After that, the crossover, mutation and LS operate on directly represented schedules, providing a means of exploring the whole search space of the layer, and not being restricted to the solutions that can be produced by the schedule builders.
- The greedy LS combined with the expanding gene pool and rotation among the long genes found in the previous layer provides a thorough examination of the search space in each layer.

Thus it can be stated that the greedy MLGLS has proven to be an efficient optimisation method.

CHAPTER 6

Research results and summary

5.6 Research results

This thesis reports on the development of a novel optimisation technique, a multi-layered genetic algorithm. The technique was demonstrated on an example of a maintenance scheduling problem.

Traditional GAs with direct representation, acting on a population of schedules, have difficulties in finding good solutions due to the overly large search space. In indirect representation, the algorithm operates on sequences of units transferred into schedules via a schedule builder procedure. This approach proved to be more successful than the traditional GA with direct representation. However, the indirect representation cannot explore the entire problem domain - rather it performs a search in a specific area defined by the schedule builder.

A novel approach was proposed as an effective search technique suitable for a large problem domain. This approach employs the idea of a partial separability of the scheduling problem by dividing the problem into layers and solving the resulting sequential problems one after another, gradually building a solution to the initial large problem. The new approach was called a multi-layered genetic algorithm; it has the following essential features:

- The problem is divided into layers, that is, several smaller problems that are solved subsequently one after another. A GA in each layer seeks partial solutions to the problem within the search space of the layer.
- A criterion is selected that identifies, which partial solutions found in a layer should be included into the next layer gene pool as complete building blocks. The aim of each layer algorithm is to find a specified number of partial solutions satisfying the criterion.
- The GA in each subsequent layer utilises partial solutions from the previous layer adding to them new components from the current layer in order to build larger solutions. Thus, each individual in subsequent layers contains a long gene representing a partial solution built in the previous layer.

The case study was investigated further and two different ways to divide the scheduling problem into layers were suggested. Some guidelines were also recommended for

general optimisation problems. Two gene pool criteria were proposed and their effect on MLGA performance was examined using several case studies. A number of features that improve the algorithm efficiency were suggested, including:

- Increasing the population size at the end of the run to speed up the search for the last few solutions in a layer.
- Reducing the number of allowed values for single genes in the beginning of a layer, if the gene pool containing partial solutions from the previous layer has partially converged.

It was found that the MLGA provided a better exploration of the problem domain when compared with the traditional GA that used the same direct representation. The MLGA was able to produce near optimal solutions; however, its performance was still inferior compared with the best performance of the GAs employing an indirect representation and a schedule builder.

Nevertheless, the proposed algorithm provided an opportunity to explore the entire problem domain more efficiently and, since the algorithm itself was rather simple and makes it possible to add other search enhancing techniques into it, further modifications of the MLGA were proposed. For example, the MLGA was combined with a local search method to improve the partial solutions found by a layer GA. To perform the local search in the MLGA the notion of a neighbourhood of a solution was introduced. The neighbourhood was defined for the maintenance scheduling problem and the essential parameters of the algorithm were identified and investigated in a number of case studies.

Incorporating a local search operator into the MLGA dramatically improved the exploration of the problem domain. The resulting algorithm, a multi-layered genetic local search, MLGLS, was up to five times faster than a MLGA and its success rate was up to 50% better than that of a MLGA, depending on the problem's constraints.

Further development of the MLGLS algorithm, a greedy MLGLS, facilitated the use of the local search for even more efficient exploration of the problem domain. The greedy MLGLS not only improved fitness of individuals within the population, but also included all good solutions found in the neighbourhoods by a local search operator, into the next layer gene pool. The new algorithm took full advantage of the MLGLS ability

to search a neighbourhood of a partial solution, and thus increased the overall algorithm efficiency.

The best results, however, were obtained when a greedy MLGLS was combined with heuristic initialisation procedures. It was demonstrated that the algorithm was then able to explore a large problem domain more efficiently, with a higher probability of finding the optimal solution.

5.7 Software development

Research carried out in this work included the formulation of the problem and designing algorithms suitable for solving it. These algorithms included a traditional GA with direct and indirect representation, as well as an MLGA, an MLGLS and a greedy MLGLS.

In order to implement various algorithms and tune their parameters, a complete software package was developed by the candidate using MATLAB programming language on a standard desktop computer. The MLGA was presented as a series of single-layer GAs using partial solutions as single genes to be built up into larger solutions. A number of specific MLGA features were implemented in the software package:

- Poolwise representation of individuals and poolwise mutation for an algorithm with direct representation.
- Representation of individuals as permutations, as well as mutation and recombination operators to be used on such individuals, if an algorithm with an indirect representation was evaluated.
- A replacement strategy that automatically weeds out duplicates.
- Two models of a schedule builder to be used in a GA with indirect representation or as an initialisation stochastic routine in a greedy MLGLS.
- A local search operator that was added into the standard GA cycle, and a special procedure describing an individual's neighbourhood. This procedure was designed to minimise the number of potential neighbours of an individual by taking into account the additional load on the system that results from scheduling units from the

individual's long gene. The evaluation of the individuals also utilised the additional load and, as a result, became faster than a similar evaluation would have been in a traditional GA.

- Age management procedure, which involved including age of an individual as its attribute.
- A procedure for weeding out interchangeable partial solutions from a gene pool with the emphasis on the fast processing of a large number of solutions.

Special care was taken to store all relevant experimental data in a format that would later enable the researcher to retrieve and process the data for evaluation of the algorithms. This data was also used in tables and figures presented in this thesis. Several specialised procedures have been designed to read the data from files and to produce tables and figures according to specifications.

5.8 Further research

Further research should be focused on investigating possible application areas for multi-layered algorithms. There is a variety of problems that could be solved using this approach, including:

- Scheduling in real time mode with emergency rescheduling.
- Strategy planning and project development, which can be done gradually over a period of time or within vaguely defined constraints concerning the components of the project.

5.9 Summary

Case studies conducted in this research show that the proposed MLGA and its subsequent development present a highly efficient and fast method of exploration of a large search space. By dividing a problem into layers, the search is focused in potentially good areas and is, as a result, more efficient, especially if combined with

some performance enhancing technique. From the perspective that GAs can be unacceptably slow when applied to many real-life problems, the new algorithm broadens the range of solvable problems.

The proposed algorithm has several advantages:

- Direct representation is employed, which ensures a complete search of the problem domain, and can to be used for the majority of problems.
- The problem is divided into layers and, as a result, the search is directed into several promising areas instead of a purely random search through the vast problem domain. However, these promising areas vary in every run, therefore the search is conducted across the entire problem domain and not just in the area determined by the heuristic schedule builders as in a GA with indirect representation.
- The algorithm employs the simple idea of gradually solving the problem and uses basic GA features and genetic operators. As a result, it is easy to understand and implement.
- Moreover, the MLGA can incorporate a variety of performance enhancing techniques, such as local search and optimisation heuristics, as was shown in this thesis. Other possible performance improving methods include parallel GAs, combination with simulated annealing or the use of fuzzy controllers for dynamic adjustment of the algorithm's parameters.

The multi-layered genetic algorithm has proven to be a powerful optimisation method.

APPENDIX 1

Gene pools for direct representation

Table A1.1 Gene pools for direct representation, R^0=0

Unit number / Gene number			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Unit capacity			150	150	150	210	210	210	210	210	210	230	160	210	210	210	210	100	100	200	200	210	100	50
Number of weeks requred			3	3	3	4	4	4	4	4	4	8	2	4	4	4	4	4	4	4	4	4	5	4
Week number and gross reserve of the system in that week	1	1410	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	1260	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	1260	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	1290	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	1310	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	1330	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	1340	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	8	1390	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	1420	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	1510	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	1580	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	12	1760	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	1970	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	2040	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	2090	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	2060	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	2120	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	2190	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	2250	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	2290	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	2340	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	2370	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	2390	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	2410	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	2390	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	2400	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	2420	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	2410	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	2400	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	2390	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	2390	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	2360	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	2350	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
	34	2320	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34
	35	2310	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
	36	2240	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
	37	2210	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
	38	2160	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
	39	2090	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39
	40	1970	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
	41	1930	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
	42	1740	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42
	43	1690	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43
	44	1540	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44
	45	1530	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45
	46	1480	46	46	46	46	46	46	46	46	46		46	46	46	46	46	46	46	46	46	46	46	46
	47	1450	47	47	47	47	47	47	47	47	47		47	47	47	47	47	47	47	47	47	47	47	47
	48	1380	48	48	48	48	48	48	48	48	48		48	48	48	48	48	48	48	48	48	48	48	48
	49	1310	49	49	49	49	49	49	49	49	49		49	49	49	49	49	49	49	49	49	49		49
	50	1320	50	50	50								50											
	51	1390											51											
	52	1480																						

Table A1.1 (continued) Gene pools for direct representation, R^0=0

Unit number / Gene number			23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Unit capacity			60	60	60	30	30	60	60	30	30	30	20	80	80	130	460	460	460	460	120	120	170
Number of weeks requred			4	4	4	7	7	7	7	3	3	3	2	1	1	3	8	11	6	4	1	1	1
Week number and gross reserve of the system in that week	1	1410	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	2	1260	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
	3	1260	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
	4	1290	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
	5	1310	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
	6	1330	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
	7	1340	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	8	1390	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
	9	1420	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	10	1510	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	1580	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	12	1760	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	1970	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	2040	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	2090	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	2060	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	2120	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	2190	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	2250	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	2290	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	2340	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	2370	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	2390	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	2410	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	2390	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	2400	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	2420	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	2410	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	2400	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	2390	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	2390	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	2360	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	2350	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
	34	2320	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34
	35	2310	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
	36	2240	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
	37	2210	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
	38	2160	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
	39	2090	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39
	40	1970	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
	41	1930	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41	41
	42	1740	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42	42
	43	1690	43	43	43	43	43	43	43	43	43	43	43	43	43	43	43		43	43	43	43	43
	44	1540	44	44	44	44	44	44	44	44	44	44	44	44	44	44	44		44	44	44	44	44
	45	1530	45	45	45	45	45	45	45	45	45	45	45	45	45	45	45		45	45	45	45	45
	46	1480	46	46	46	46	46	46	46	46	46	46	46	46	46	46			46	46	46	46	46
	47	1450	47	47	47					47	47	47	47	47	47	47			47	47	47	47	47
	48	1380	48	48	48					48	48	48	48	48	48	48				48	48	48	48
	49	1310	49	49	49					49	49	49	49	49	49	49				49	49	49	49
	50	1320								50	50	50	50	50	50	50					50	50	50
	51	1390											51	51	51						51	51	51
	52	1480												52	52						52	52	52

Table A1.2 Gene pools for direct representation, R^0=1220

Unit number / Gene number			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Unit capacity			150	150	150	210	210	210	210	210	210	230	160	210	210	210	210	100	100	200	200	210	100	50
Number of weeks requred			3	3	3	4	4	4	4	4	4	8	2	4	4	4	4	4	4	4	4	4	5	4
Week number and gross reserve of the system in that week minus 1220	1	190																						
	2	40																						
	3	40																						
	4	70																						4
	5	90																						5
	6	110																6	6				6	6
	7	120																7	7				7	7
	8	170	8	8	8								8					8	8				8	8
	9	200	9	9	9								9					9	9	9	9		9	9
	10	290	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	360	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	12	540	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	750	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	820	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	870	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	840	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	900	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	970	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	1030	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	1070	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	1120	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	1150	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	1170	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	1190	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	1170	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	1180	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	1200	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	1190	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	1180	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	1170	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	1170	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	1140	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	1130	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
	34	1100	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34
	35	1090	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
	36	1020	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
	37	990	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
	38	940	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
	39	870	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39	39
	40	750	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40	40
	41	710	41	41	41	41	41	41	41	41	41		41	41	41	41	41	41	41	41	41	41	41	41
	42	520	42	42	42	42	42	42	42	42	42		42	42	42	42	42	42	42	42	42	42	42	42
	43	470	43	43	43	43	43	43	43	43	43		43	43	43	43	43	43	43	43	43	43	43	43
	44	320	44	44	44	44	44	44	44	44	44		44	44	44	44	44	44	44	44	44	44	44	44
	45	310	45	45	45								45					45	45					45
	46	260	46	46	46								46											46
	47	230											47											47
	48	160																						48
	49	90																						49
	50	100																						
	51	170											51											
	52	260																						

Table A1.2 (continued) Gene pools for direct representation, R^0=1220

Unit number / Gene number			23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Unit capacity			60	60	60	30	30	60	60	30	30	30	20	80	80	130	460	460	460	460	120	120	170
Number of weeks requred			4	4	4	7	7	7	7	3	3	3	2	1	1	3	8	11	6	4	1	1	1
Week number and gross reserve of the system in that week minus 1220	1	190				1	1			1	1	1	1	1	1						1	1	1
	2	40				2	2			2	2	2	2										
	3	40				3	3			3	3	3	3										
	4	70	4	4	4	4	4	4	4	4	4	4	4										
	5	90	5	5	5	5	5	5	5	5	5	5	5	5	5								
	6	110	6	6	6	6	6	6	6	6	6	6	6	6	6								
	7	120	7	7	7	7	7	7	7	7	7	7	7	7	7						7	7	
	8	170	8	8	8	8	8	8	8	8	8	8	8	8	8	8					8	8	8
	9	200	9	9	9	9	9	9	9	9	9	9	9	9	9	9					9	9	9
	10	290	10	10	10	10	10	10	10	10	10	10	10	10	10	10					10	10	10
	11	360	11	11	11	11	11	11	11	11	11	11	11	11	11	11					11	11	11
	12	540	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	750	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	820	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	870	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	840	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	900	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	970	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	1030	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	1070	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	1120	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	1150	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	1170	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	1190	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	1170	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	1180	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	1200	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	1190	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	1180	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	1170	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	1170	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	1140	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	1130	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
	34	1100	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34		34	34	34	34	34
	35	1090	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35		35	35	35	35	35
	36	1020	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36		36	36	36	36	36
	37	990	37	37	37	37	37	37	37	37	37	37	37	37	37	37			37	37	37	37	37
	38	940	38	38	38	38	38	38	38	38	38	38	38	38	38	38			38	38	38	38	38
	39	870	39	39	39	39	39	39	39	39	39	39	39	39	39	39				39	39	39	39
	40	750	40	40	40	40	40	40	40	40	40	40	40	40	40	40				40	40	40	40
	41	710	41	41	41	41	41	41	41	41	41	41	41	41	41	41					41	41	41
	42	520	42	42	42	42	42	42	42	42	42	42	42	42	42	42					42	42	42
	43	470	43	43	43	43	43	43	43	43	43	43	43	43	43	43					43	43	43
	44	320	44	44	44	44	44	44	44	44	44	44	44	44	44	44					44	44	44
	45	310	45	45	45	45	45	45	45	45	45	45	45	45	45	45					45	45	45
	46	260	46	46	46	46	46	46	46	46	46	46	46	46	46	46					46	46	46
	47	230	47	47	47					47	47	47	47	47	47						47	47	47
	48	160	48	48	48					48	48	48	48	48	48						48	48	
	49	90	49	49	49					49	49	49	49	49	49								
	50	100								50	50	50	50	50	50								
	51	170											51	51	51						51	51	51
	52	260												52	52						52	52	52

Table A1.3 Gene pools for direct representation, R^0=1260

Unit number / Gene number			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Unit capacity			150	150	150	210	210	210	210	210	210	230	160	210	210	210	210	100	100	200	200	210	100	50
Number of weeks requred			3	3	3	4	4	4	4	4	4	8	2	4	4	4	4	4	4	4	4	4	5	4
Week number and gross reserve of the system in that week minus 1260	1	150																						
	2	0																						
	3	0																						
	4	30																						
	5	50																						5
	6	70																						6
	7	80																						7
	8	130																8	8				8	8
	9	160	9	9	9								9					9	9				9	9
	10	250	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10	10
	11	320	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
	12	500	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	710	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	780	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	830	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	800	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	860	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	930	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	990	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	1030	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	1080	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	1110	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	1130	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	1150	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	1130	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	1140	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	1160	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	1150	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	1140	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	1130	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	1130	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	1100	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	1090	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33
	34	1060	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34
	35	1050	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35
	36	980	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36	36
	37	950	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37	37
	38	900	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38	38
	39	830	39	39	39	39	39	39	39	39	39		39	39	39	39	39	39	39	39	39	39	39	39
	40	710	40	40	40	40	40	40	40	40	40		40	40	40	40	40	40	40	40	40	40	40	40
	41	670	41	41	41	41	41	41	41	41	41		41	41	41	41	41	41	41	41	41	41	41	41
	42	480	42	42	42	42	42	42	42	42	42		42	42	42	42	42	42	42	42	42	42	42	42
	43	430	43	43	43	43	43	43	43	43	43		43	43	43	43	43	43	43	43	43	43	43	43
	44	280	44	44	44								44					44	44				44	44
	45	270	45	45	45								45					45	45					45
	46	220											46											46
	47	190																						47
	48	120																						48
	49	50																						49
	50	60																						
	51	130																						
	52	220																						

Table A1.3 (continued) Gene pools for direct representation, R^0=1260

Unit number / Gene number			23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43
Unit capacity			60	60	60	30	30	60	60	30	30	30	20	80	80	130	460	460	460	460	120	120	170
Number of weeks requred			4	4	4	7	7	7	7	3	3	3	2	1	1	3	8	11	6	4	1	1	1
Week number and gross reserve of the system in that week minus 1260	1	150												1	1						1	1	
	2	0																					
	3	0																					
	4	30				4	4			4	4	4	4										
	5	50				5	5			5	5	5	5										
	6	70	6	6	6	6	6	6	6	6	6	6	6										
	7	80	7	7	7	7	7	7	7	7	7	7	7	7	7								
	8	130	8	8	8	8	8	8	8	8	8	8	8	8	8	8					8	8	
	9	160	9	9	9	9	9	9	9	9	9	9	9	9	9	9					9	9	
	10	250	10	10	10	10	10	10	10	10	10	10	10	10	10	10					10	10	10
	11	320	11	11	11	11	11	11	11	11	11	11	11	11	11	11					11	11	11
	12	500	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12
	13	710	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13
	14	780	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14	14
	15	830	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15	15
	16	800	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
	17	860	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17	17
	18	930	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	19	990	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19	19
	20	1030	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20
	21	1080	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
	22	1110	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22	22
	23	1130	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23	23
	24	1150	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24	24
	25	1130	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25	25
	26	1140	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26	26
	27	1160	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27	27
	28	1150	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28	28
	29	1140	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29	29
	30	1130	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30	30
	31	1130	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31	31
	32	1100	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32	32
	33	1090	33	33	33	33	33	33	33	33	33	33	33	33	33	33	33		33	33	33	33	33
	34	1060	34	34	34	34	34	34	34	34	34	34	34	34	34	34	34		34	34	34	34	34
	35	1050	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35		35	35	35	35	35
	36	980	36	36	36	36	36	36	36	36	36	36	36	36	36	36			36	36	36	36	36
	37	950	37	37	37	37	37	37	37	37	37	37	37	37	37	37			37	37	37	37	37
	38	900	38	38	38	38	38	38	38	38	38	38	38	38	38	38				38	38	38	38
	39	830	39	39	39	39	39	39	39	39	39	39	39	39	39	39				39	39	39	39
	40	710	40	40	40	40	40	40	40	40	40	40	40	40	40	40					40	40	40
	41	670	41	41	41	41	41	41	41	41	41	41	41	41	41	41					41	41	41
	42	480	42	42	42	42	42	42	42	42	42	42	42	42	42	42					42	42	42
	43	430	43	43	43	43	43			43	43	43	43	43	43	43					43	43	43
	44	280	44	44	44	44	44			44	44	44	44	44	44	44					44	44	44
	45	270	45	45	45	45	45			45	45	45	45	45	45	45					45	45	45
	46	220				46	46			46	46	46	46	46	46						46	46	46
	47	190								47	47	47	47	47	47						47	47	47
	48	120								48	48	48	48	48	48						48	48	
	49	50								49	49	49	49										
	50	60								50	50	50	50										
	51	130											51	51	51						51	51	
	52	220												52	52						52	52	52

APPENDIX 2

Performance graphs for a 12-layer MLGA

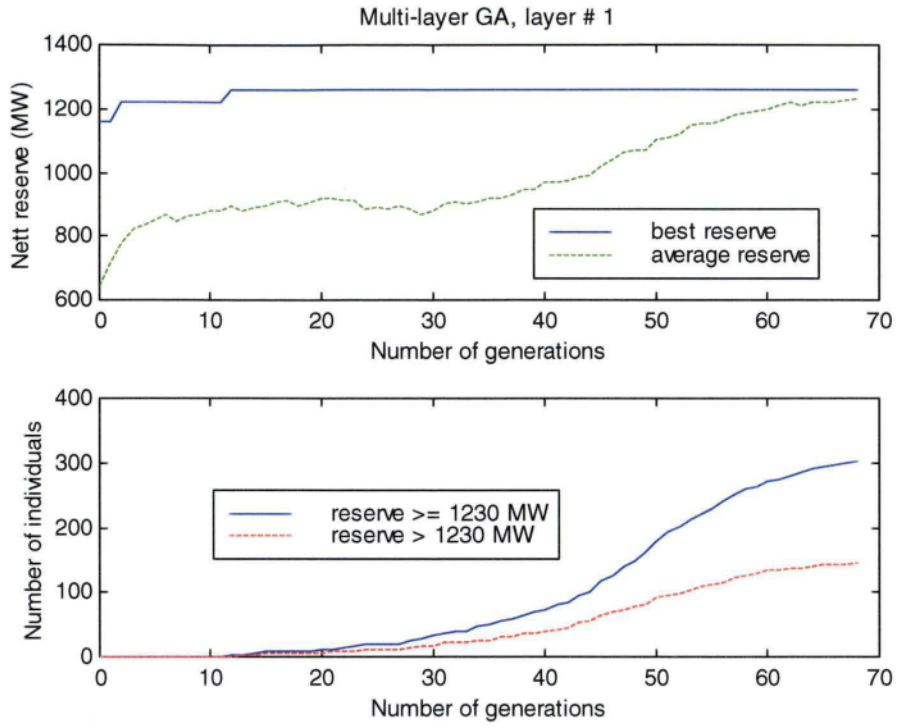

Figure A2.1 Performance graph for the 1st layer of MLGA

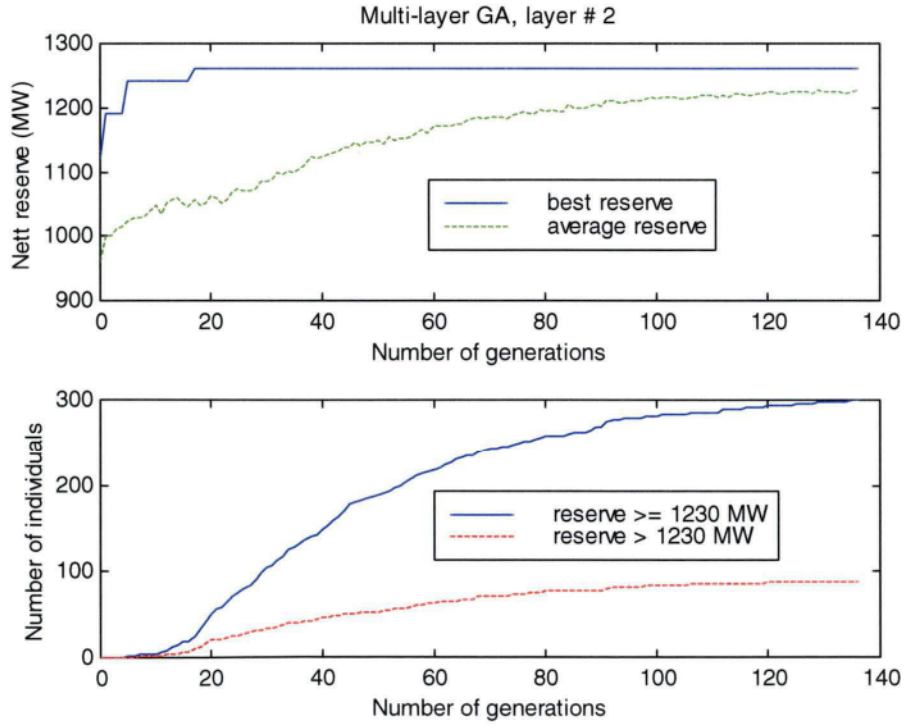

Figure A2.2 Performance grapf of the 2nd layer of MLGA

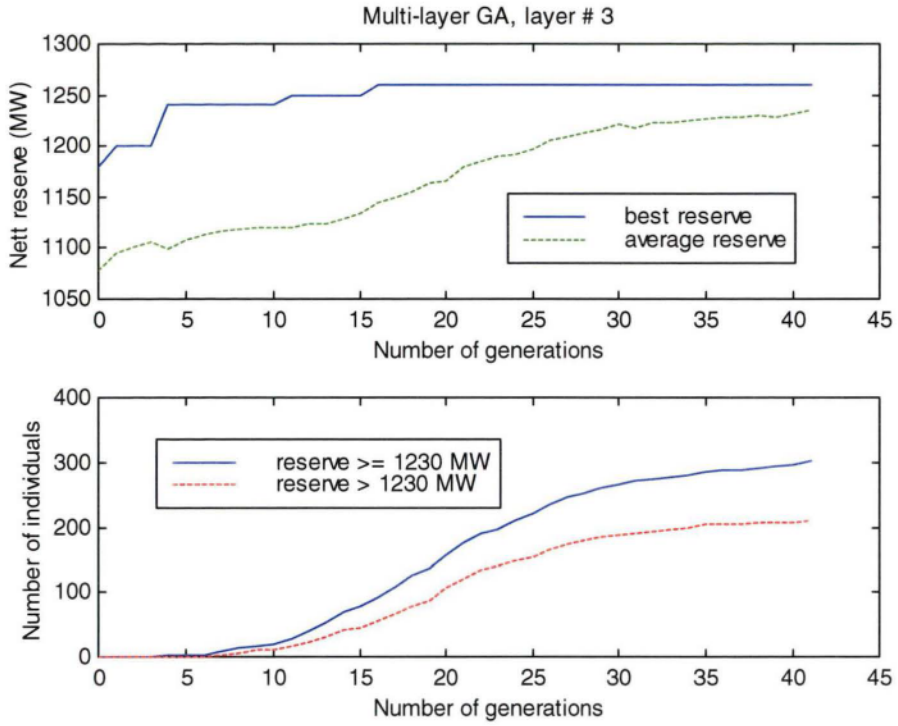

Figure A2.3 Performance graph for the 3d layer of MLGA

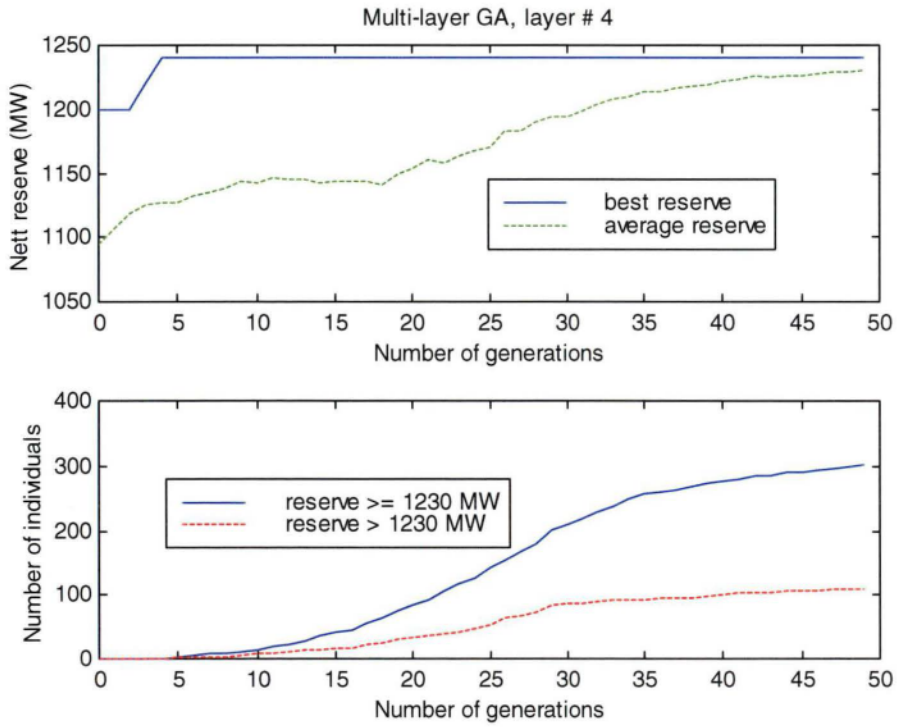

Figure A2.4 Performance graph for the 4th layer of MLGA

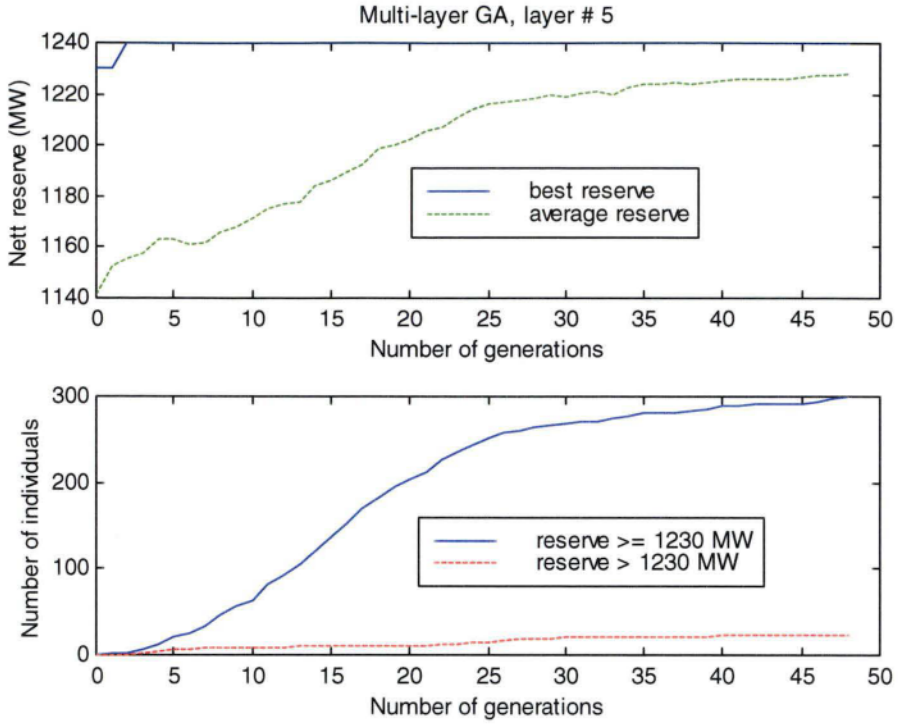

Figure A2.5 Performance graph for the 5th layer of MLGA

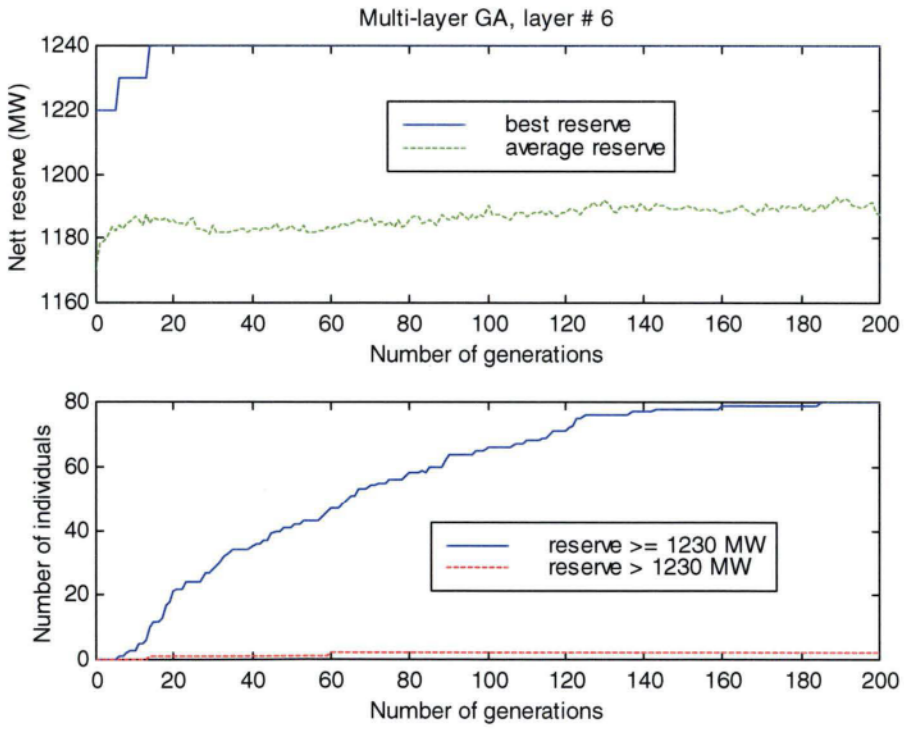

Figure A2.6 Performance graph for the 6th layer of MLGA

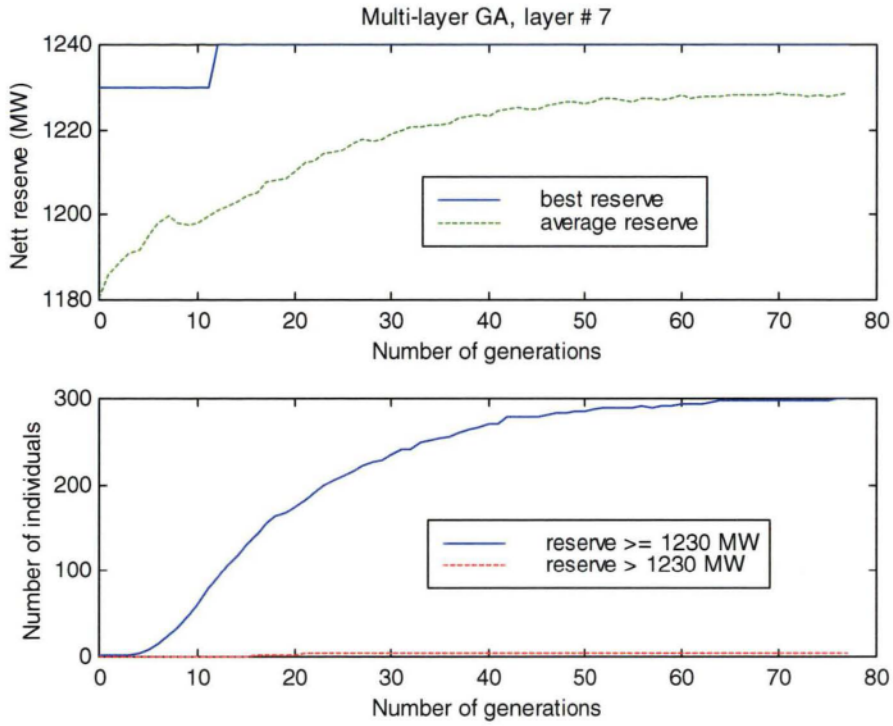

Figure A2.7 Performance graph for the 7th layer of MLGA

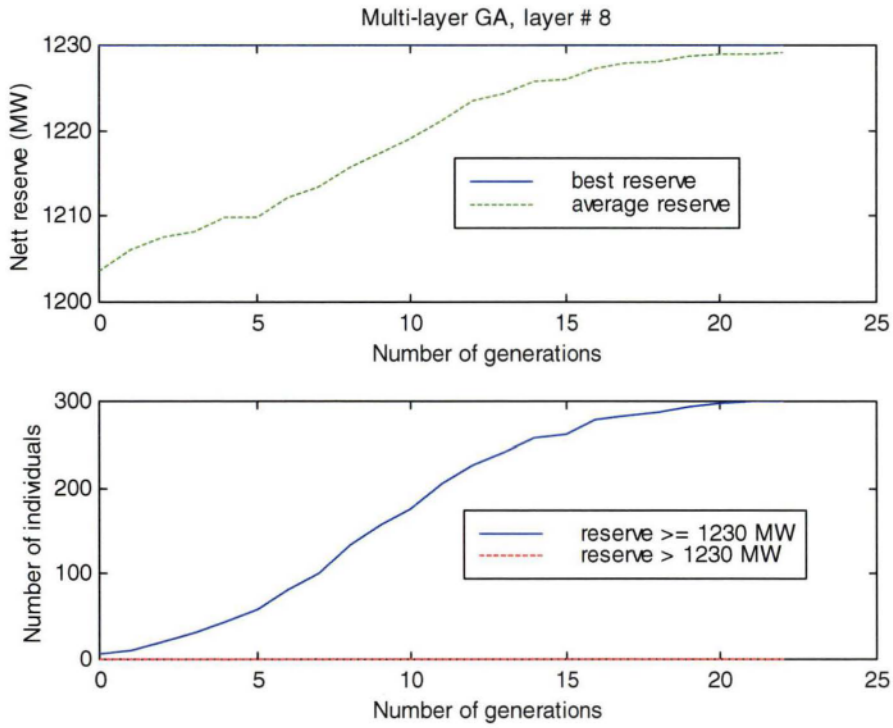

Figure A2.8 Performance graph for the 8th layer of MLGA

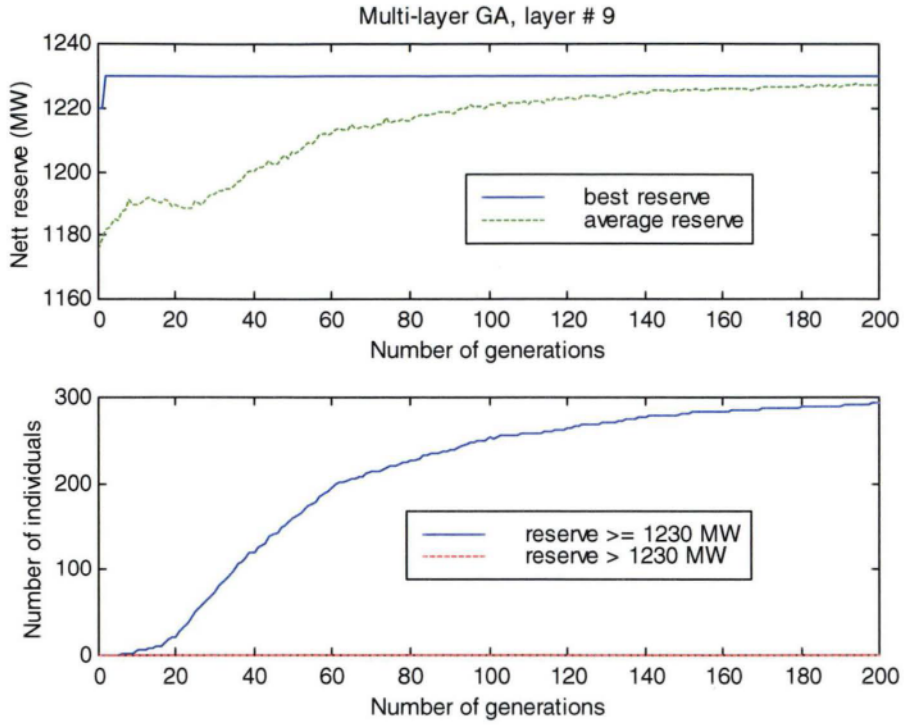

Figure A2.9Performance graph for the 9th layer of MLGA

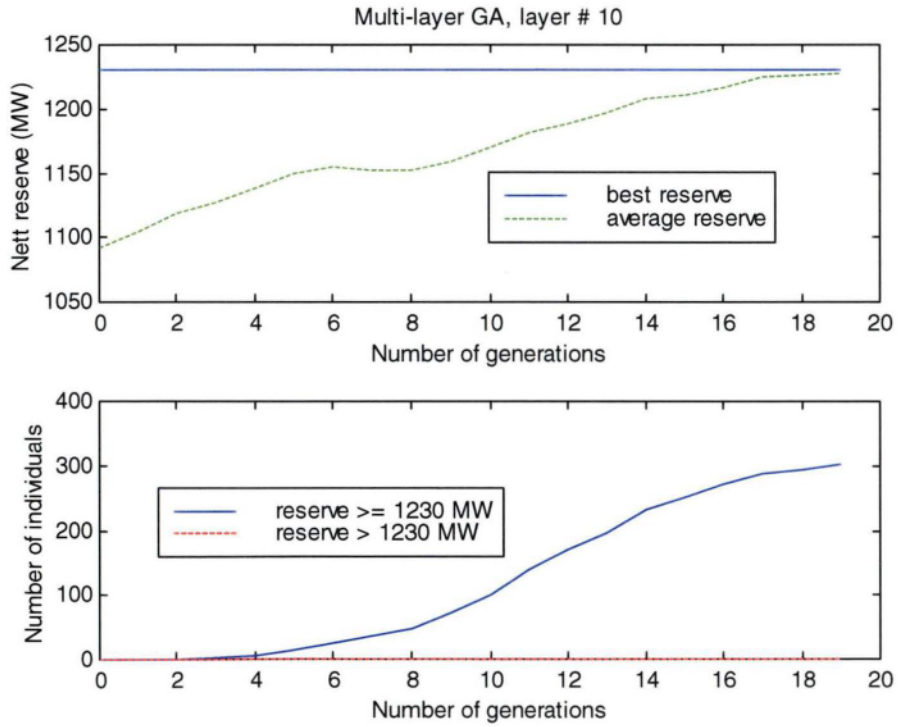

Figure A2.10 Performance graph for the 10th layer of MLGA

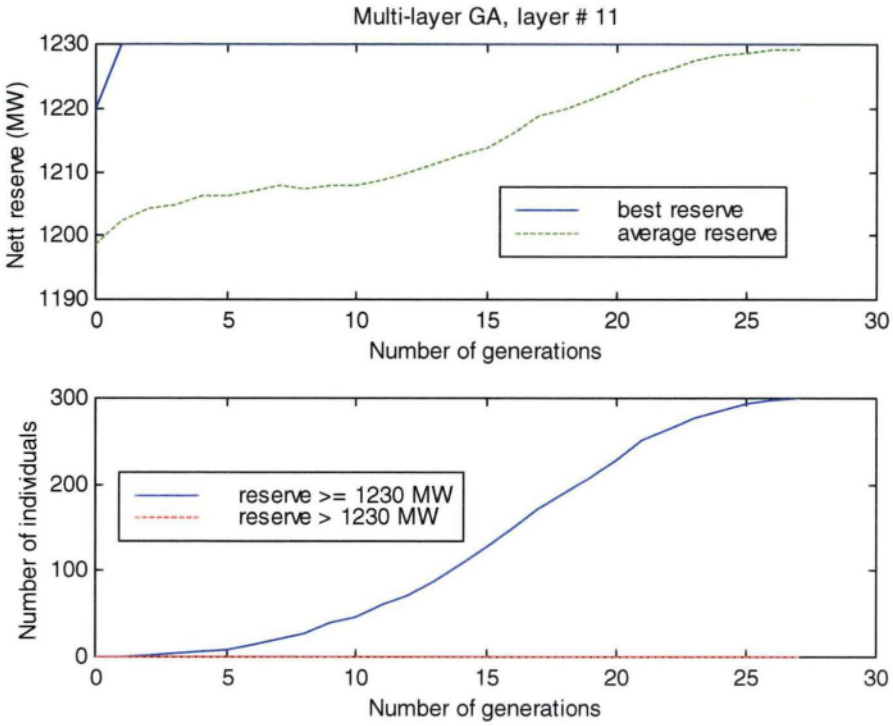

Figure A2.11 Performance graph for the 11th layer of MLGA

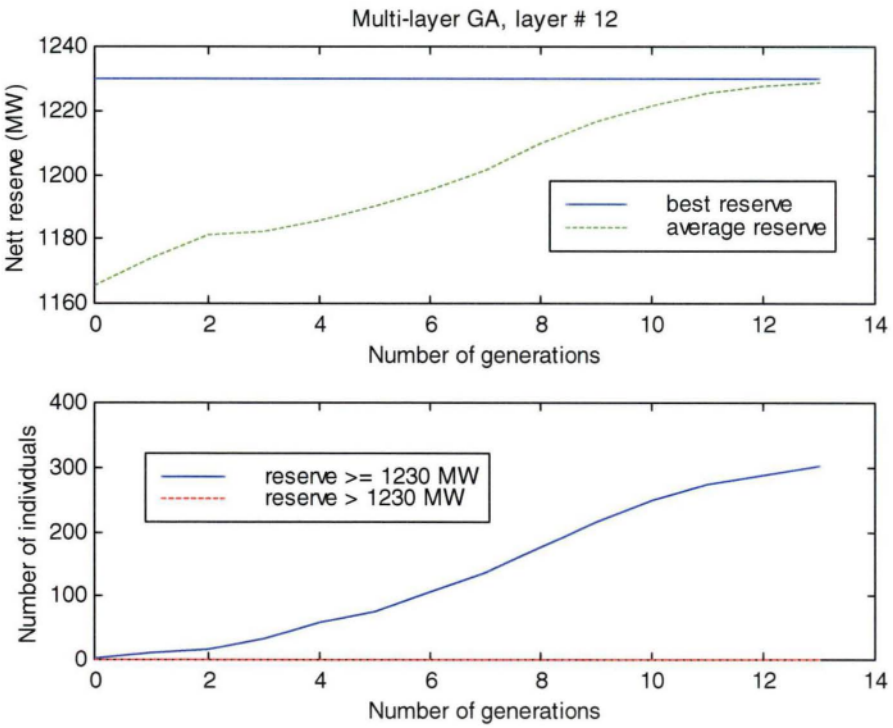

Figure A2.12 Performance graph for the 12th layer of MLGA

APPENDIX 3

Layered results for a 12-layer MLGA

Table A3.1 Layered results for a 12-layer GA. Pool threshold = 1220.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged units at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	49.97	50.25	1133.25	640.93	4.53	1260	1260	1222.33	299.15	0
2	40	96.71	99.60	1183.75	968.59	3.87	1254.12	1251.50	1213.40	250.10	3.20
3	40	32.56	33.13	1221.00	1077.30	1.33	1241.47	1239.50	1218.88	267.83	6
4	40	44.12	44.78	1213.75	1090.21	1.53	1231.47	1230.50	1217.73	301.23	9.80
5	40	66.29	64.88	1221.50	1131.73	1.20	1227.35	1227.00	1217.63	242.00	14.20
6	40	93.41	100.40	1214.50	1159.11	2.07	1223.24	1223.00	1211.75	195.15	15.60
7	40	48.82	52.83	1220	1167.16	1	1221.47	1221.25	1215.40	237.85	16.40
8	40	39.44	47.03	1218.25	1189.85	1.67	1221.18	1221.00	1218.58	253.85	17.20
9	34	132.76	130.35	1211.00	1165.84	2.57	1220.59	1219.00	1203.13	233.21	24.80
10	34	17.59	17.59	1212.35	1087.16	1.50	1220	1220	1216.62	292.79	25.20
11	34	24.06	24.06	1218.24	1190.06	1.43	1220	1220	1219.12	279.97	26.40
12	34	10.91	10.91	1220	1160.34	1	1220	1220	1218.68	303.71	26.80

Table A3.2 Layered results for a 12-layer GA. Pool threshold = 1230.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged units at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	62.07	59.45	1154.50	647.77	5.90	1260	1260	1230.98	299.53	0
2	40	134.44	127.03	1198.50	975.87	2.60	1250	1251.25	1220.08	231.63	3.30
3	40	33.07	33.10	1230.75	1083.41	1.10	1243.70	1242.50	1228.75	261.98	5.90
4	40	43.96	45.85	1221.75	1096.08	2.60	1236.67	1237.00	1226.80	301.28	11.10
5	40	76.07	81.65	1228.50	1140.65	1.10	1233.33	1233.25	1225.98	217.95	12.60
6	37	151.33	169.73	1219.50	1168.33	3.00	1231.48	1230.25	1205.63	124.14	15.20
7	34	70.67	101.35	1226.76	1175.36	1.56	1231.11	1230.00	1219.68	206.53	16.78
8	34	32.89	54.03	1228.24	1199.47	1.67	1230	1230	1227.06	238.41	19.67
9	27	170.93	176.91	1215.59	1173.88	2.83	1230	1226.47	1206.41	191.04	25.33
10	27	28.56	28.56	1212.22	1092.59	2.33	1230	1230	1224.59	278.93	25.33
11	27	27.93	27.93	1225.93	1198.61	2	1230	1230	1228.96	269.85	26.17
12	27	13.30	13.30	1230	1165.99	1	1230	1230	1228.74	302.63	25.33

Table A3.3 Layered results for a 12-layer GA. Pool threshold = 1240.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged units at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	67.00	57.05	1166.50	652.84	6.10	1260	1260	1233.73	298.78	0
2	40	245.00	182.55	1189.50	976.81	7.37	1260	1255.25	1208.68	207.63	2.20
3	40	35.00	43.85	1231.50	1087.38	1.87	1250	1248.00	1234.63	225.48	4.95
4	40	60.00	123.80	1216.00	1096.62	5.90	1250	1243.75	1229.30	287.85	10.15
5	36	70.00	199.90	1224.75	1142.99	8.30	1245	1239.75	1196.78	114.61	13.53
6	21	100.00	187.92	1217.78	1173.11	13.24	1240	1230.56	1195.22	77.71	16.92
7	18	46.00	149.29	1232.86	1180.91	2.67	1240	1238.10	1216.29	127.78	18.18
8	11	113.50	183.72	1230.00	1206.48	8.00	1240	1236.11	1223.44	128.00	22.50
9	2	290.00	216.36	1213.64	1180.59	11.00	1240	1220.91	1190.64	21.50	24.00
10	2	33.50	33.50	1225.00	1095.37	3.00	1240	1240	1234.50	243.50	24.00
11	2	48.50	48.50	1230.00	1206.40	2.00	1240	1240	1238.50	213.50	29.00
12	2	16.00	16.00	1240	1170.84	1	1240	1240	1238.00	298.00	29.00

Table A3.4 Layered results for a 12-layer GA. Pool threshold = 1250.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged units at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	61.10	1140.75	645.24	8.33	n/a	1260	1238.55	298.20	0
2	39	n/a	204.35	1195.00	982.78	19.90	n/a	1258.50	1166.88	141.85	3.15
3	39	n/a	83.82	1236.15	1088.37	4.31	n/a	1258.21	1232.08	161.87	5.23
4	39	n/a	211.67	1205.64	1102.50	17.41	n/a	1255.38	1194.10	167.90	13.23
5	19	n/a	215.90	1217.95	1151.35	25.47	n/a	1238.72	1172.97	31.47	17.47
6	0	n/a	200.00	1222.11	1183.75	0.00	n/a	1228.42	1195.26	0.00	0.00

Table A3.5 Layered results for a 12-layer GA. Pool threshold = 1260.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged units at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	65.10	1156.34	643.76	8.76	n/a	1260	1243.55	293.34	0
2	39	n/a	211.27	1189.67	987.33	21.23	n/a	1259.75	1179.44	123.75	4.34
3	38	n/a	99.87	1234.32	1076.35	5.98	n/a	1259.50	1239.45	152.65	7.83
4	29	n/a	207.33	1210.47	1112.11	20.56	n/a	1247.48	1203.45	145.21	13.99
5	7	n/a	240.54	1223.65	11.67.45	30.66	n/a	1234.56	1188.56	24.32	18.32
6	0	n/a	200.00	1220.33	1192.47	0.00	n/a	1231.43	1201.56	0.00	0.00

Table A3.6 Layered results for a 12-layer GA with added selection pressure. Pool threshold = 1220.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	52.95	52.93	1142.25	644.96	5.30	1260	1260	1219.04	299.18	0
2	40	67.59	66.18	1211.25	973.05	2.08	1248	1247.25	1214.68	274.98	2.00
3	40	25.00	25.48	1224.50	1077.01	1.08	1242	1241.25	1218.33	272.93	5.40
4	40	30.89	31.48	1220.75	1090.86	1.13	1235	1234.50	1216.57	302.35	9.60
5	40	40.14	45.30	1222.25	1133.70	1.10	1228	1228.50	1216.03	251.75	13.00
6	40	76.65	77.60	1218.75	1159.22	1.30	1224	1224.75	1215.36	215.93	15.00
7	40	44.59	48.73	1219.75	1169.05	1.35	1224	1224	1217.03	258.18	17.70
8	40	25.89	34.43	1220.25	1192.68	1.10	1222	1222	1218.11	270.83	20.90
9	37	85.65	94.23	1216.50	1166.92	1.65	1221	1220.25	1208.29	255.08	23.60
10	37	12.70	12.70	1219.73	1107.18	1.03	1221	1221	1215.94	295.14	23.70
11	37	15.81	15.81	1219.46	1190.85	1	1220	1220	1218.68	294.49	24.40
12	37	8.57	8.57	1220	1169.29	1	1220	1220	1218.35	305.43	25.50

Table A3.7 Layered results for a 12-layer GA with added selection pressure. Pool threshold = 1230.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	57.93	60.00	1144.50	646.44	6.15	1260	1260	1226.97	299.60	0
2	40	82.68	94.08	1215.50	981.24	2.10	1253.57	1253.75	1221.78	256.65	2.20
3	40	35.61	38.50	1230.25	1084.30	1.28	1247.86	1246.25	1226.12	248.85	5.40
4	40	44.29	49.45	1220.00	1096.14	2.45	1238.93	1238.00	1224.41	301.08	9.60
5	40	68.36	99.15	1227.25	1140.15	1.53	1233.93	1233.50	1215.82	197.53	12.70
6	38	81.00	109.80	1224.25	1166.66	2.82	1231.07	1230.25	1214.59	168.83	16.65
7	38	49.57	75.89	1228.95	1173.80	1.16	1230	1231	1219.76	198.03	18.45
8	34	41.39	71.74	1228.68	1199.64	2.32	1230	1229	1224.38	201.88	19.41
9	28	161.79	168.53	1221.76	1173.04	3.18	1230	1227.35	1206.91	141.03	22.07
10	28	14.79	14.79	1226.43	1110.50	1.14	1230	1230	1225.27	196.58	23.93
11	28	23.43	23.43	1228.57	1197.83	2	1230	1230	1228.59	189.43	25.07
12	28	10.04	10.04	1230	1175.78	1	1230	1230	1228.24	213.68	26.57

Table A3.8 Layered results for a 12-layer GA with added selection pressure. Pool threshold = 1240.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	63.57	58.78	1142.25	642.20	5.75	1260	1260	1230.47	297.88	0
2	40	89.57	110.50	1213.00	984.35	2.45	1254	1254.25	1225.06	247.58	3.25
3	40	32.00	45.90	1233.25	1086.36	1.58	1247	1248.00	1233.80	231.38	6.50
4	40	63.00	105.23	1221.25	1096.88	3.40	1244	1243.25	1230.04	292.20	11.90
5	40	177.86	194.80	1231.75	1141.79	2.58	1241	1240.75	1196.49	102.50	15.10
6	31	151.29	181.28	1226.00	1171.72	7.81	1240	1235.75	1198.70	74.29	18.65
7	26	84.43	151.42	1235.16	1180.35	1.32	1240	1238	1211.48	105.08	20.36
8	14	140.14	177.73	1233.08	1204.41	1.70	1240	1237	1218.01	86.07	24.75
9	7	180.00	180.00	1227.86	1178.30	1.86	1240	1231.43	1195.66	46.29	25.33
10	7	12.86	12.86	1240.00	1135.55	1.00	1240	1240	1235.66	257.14	27.33
11	7	73.29	73.29	1231.43	1204.84	4	1240	1240	1236.96	166.14	28.67
12	7	11.29	11.29	1239	1179.53	1	1240	1240	1237.49	300.14	28.67

Table A3.9 Layered results for a 12-layer GA with added selection pressure. Pool threshold = 1250.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	72.23	1139.25	642.77	8.43	n/a	1260	1240.81	295.08	0.8
2	40	n/a	163.23	1211.75	987.01	5.33	n/a	1259.75	1223.24	223.73	4.6
3	40	n/a	81.43	1234.75	1091.90	4.45	n/a	1258.50	1242.15	186.23	8.5
4	39	n/a	184.33	1218.25	1101.59	7.00	n/a	1256.00	1215.08	229.18	13.4
5	31	n/a	197.44	1227.44	1149.05	9.40	n/a	1246.92	1175.19	36.77	19.4
6	6	n/a	137.14	1220.95	1182.17	3.00	n/a	1229.52	1196.26	47.00	20.5
7	5	n/a	143.33	1245.00	1193.19	1.67	n/a	1250.00	1201.54	16.40	28
8	1	n/a	92.00	1238.00	1214.84	1	n/a	1238.00	1218.51	2	29
9	0	n/a	20.00	1220.00	1176.94	0	n/a	1220	1180.41	0	0

Table A3.10 Layered results for a 12-layer GA with added selection pressure. Pool threshold = 1260.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	76.63	1159.75	640.14	8.90	n/a	1260	1243.25	292.80	0.6
2	40	n/a	169.60	1220.75	985.19	4.68	n/a	1260.00	1221.65	198.68	4.7
3	40	n/a	88.33	1237.25	1094.49	2.95	n/a	1260.00	1247.91	162.35	8.8
4	39	n/a	197.18	1221.50	1105.17	9.53	n/a	1260.00	1196.07	160.24	11.7
5	11	n/a	176.32	1228.42	1154.12	12.43	n/a	1246.05	1172.89	14.55	18.7
6	0	n/a	87.27	1220.91	1190.71	0.00	n/a	1226.36	1197.77	0.00	0.0

APPENDIX 4

Layered results for a 9-layer MLGA

Table 4.1 Layered results for a 9-layer GA. Pool threshold = 1220.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	73.61	74.20	1086.75	574.72	15.80	1255.83	1256.25	1212.50	294.60	0.05
2	40	42.69	42.50	1217.75	1034.60	1.40	1244.17	1243.75	1216.50	277.45	1.20
3	40	37.44	38.03	1213.00	1074.56	1.58	1238.33	1237.75	1215.10	247.93	4.05
4	40	83.94	87.35	1193.75	1106.26	11.28	1225.00	1225.00	1213.93	247.70	12.55
5	39	119.39	127.45	1212.00	1090.36	2.00	1223.61	1223.00	1198.05	265.49	12.40
6	39	94.25	99.05	1210.51	1154.03	7.15	1220.56	1220.51	1214.95	199.31	14.50
7	39	73.22	84.00	1216.67	1189.22	6.21	1220.56	1220.51	1215.00	207.08	17.20
8	36	36.92	49.46	1189.74	1082.98	6.58	1220.00	1218.21	1210.05	290.39	18.25
9	36	35.00	35.00	1209.17	1153.75	10.50	1220.00	1220.00	1217.94	300.33	23.55

Table 4.2 Layered results for a 9-layer GA. Pool threshold = 1230.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	82.93	80.70	1077.75	571.14	18.20	1257.04	1257.25	1217.80	293.30	0.15
2	40	51.44	56.00	1224.00	1039.13	1.55	1245.93	1246.25	1224.65	270.78	1.73
3	40	50.19	56.40	1217.50	1079.66	2.43	1238.15	1238.75	1225.30	223.53	4.35
4	39	117.93	139.58	1201.50	1114.26	16.90	1232.22	1232.25	1217.25	213.95	13.49
5	38	135.96	157.21	1208.72	1098.98	4.26	1231.11	1230.51	1196.23	230.13	16.24
6	34	149.70	161.53	1213.16	1162.84	8.06	1230.37	1228.95	1214.97	137.74	19.38
7	29	65.41	92.29	1224.41	1198.04	4.07	1230	1227.94	1222.12	212.31	20.59
8	27	75.67	84.24	1195.52	1089.67	14.26	1230	1228.97	1219.07	272.85	24.00
9	27	37.67	37.67	1211.48	1160.75	9.30	1230	1230.00	1227.26	295.70	26.44

Table 4.3 Layered results for a 9-layer GA. Pool threshold = 1240.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	104.50	86.68	1094.50	568.82	21.80	1260	1260.00	1228.50	291.43	0.38
2	40	52.50	76.58	1225.00	1044.55	2.55	1240	1254.25	1231.58	259.20	2.10
3	40	28.00	83.03	1213.00	1084.92	3.28	1240	1248.50	1230.70	180.28	5.60
4	31	110.00	213.28	1198.50	1117.15	37.97	1240	1236.75	1191.78	127.71	14.97
5	19	143.00	199.55	1193.87	1099.45	26.89	1240	1230.65	1137.61	58.63	20.21
6	10	128.00	202.47	1218.95	1168.13	21.30	1240	1232.63	1197.53	63.00	22.50
7	3	126.00	185.20	1227.00	1205.58	1.67	1240	1232.00	1215.10	100.67	23.67
8	2	71.00	114.00	1193.33	1091.59	6.00	1240	1233.33	1209.67	279.00	27.00
9	2	125.50	125.50	1220.00	1163.19	42.50	1240	1240.00	1236.00	299.00	33.00

Table 4.4 Layered results for a 9-layer GA. Pool threshold = 1250.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	97.45	1087.50	562.44	31.28	n/a	1260.00	1235.93	286.53	0.35
2	40	n/a	108.45	1228.50	1046.29	3.98	n/a	1259.75	1240.55	239.18	3.85
3	40	n/a	127.75	1211.25	1089.05	6.60	n/a	1259.00	1234.73	141.30	7.85
4	14	n/a	226.30	1208.25	1124.50	62.86	n/a	1239.75	1171.75	54.36	19.86
5	1	n/a	204.29	1189.29	1105.97	11.00	n/a	1215.00	1132.71	15.00	0.00
6	1	n/a	240.00	1210.00	1179.28	198.00	n/a	1250.00	1192.00	2.00	0.00
7	0	n/a	200.00	1230.00	1217.88	0.00	n/a	1230.00	1223.00	0.00	0.00

Table 4.5 Layered results for a 9-layer GA. Pool threshold = 1260.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	115.80	1091.75	558.70	41.65	n/a	1260.00	1241.13	283.75	1.06
2	40	n/a	161.45	1223.75	1046.13	8.13	n/a	1260.00	1238.20	213.10	4.09
3	36	n/a	151.80	1219.25	1089.02	9.46	n/a	1259.25	1221.75	113.97	8.57
4	3	n/a	200.56	1205.83	1125.29	104.67	n/a	1237.50	1162.17	2.67	21.67
5	0	n/a	200.00	1206.67	1111.21	0.00	n/a	1216.67	1125.00	0.00	0.00

Table 4.6 Layered results for a 9-layer GA with added selection pressure. Pool threshold = 1220.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	74.95	74.95	1089.50	575.05	18.10	1257.25	1257.25	1213.36	295.58	1.30
2	40	40.10	40.10	1225.00	1036.00	1.10	1245.50	1245.50	1217.11	284.05	4.20
3	40	37.80	37.80	1217.25	1074.00	1.33	1235.25	1235.25	1215.45	255.08	5.10
4	40	75.48	75.48	1202.25	1107.55	5.75	1224.00	1224.00	1215.30	250.38	13.20
5	40	61.03	61.03	1218.00	1093.33	1.10	1223.00	1223.00	1212.99	298.33	15.80
6	40	69.05	69.05	1212.50	1152.61	6.50	1221.25	1221.25	1215.20	211.58	19.70
7	40	35.33	35.33	1219.75	1189.37	1.05	1221.00	1221.00	1216.90	246.23	22.00
8	40	25.23	25.23	1212.25	1100.85	1.93	1220.25	1220.25	1215.92	296.63	22.70
9	40	23.40	23.40	1213.00	1159.86	2.60	1220.00	1220.00	1218.01	300.43	25.40

Table 4.7 Layered results for a 9-layer GA with added selection pressure. Pool threshold = 1230.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	74.39	74.88	1096.00	568.04	17.73	1259.64	1258.75	1219.96	292.30	0.60
2	40	51.14	55.43	1226.00	1037.91	1.85	1251.79	1251.75	1224.64	268.45	4.40
3	40	50.68	61.98	1218.75	1079.91	2.68	1242.14	1243.00	1225.46	216.68	8.33
4	34	143.21	159.80	1200.75	1112.03	22.85	1232.50	1232.25	1202.68	178.79	16.79
5	34	102.57	109.94	1227.35	1100.34	1.12	1231.07	1231.18	1203.00	252.30	18.29
6	32	131.25	140.09	1212.65	1157.83	9.91	1230.71	1230.00	1213.54	115.66	20.08
7	28	76.50	86.31	1228.13	1199.36	1.10	1230	1228.75	1220.79	191.71	22.91
8	28	29.71	29.71	1218.93	1108.65	2.50	1230	1230.00	1225.49	288.57	24.45
9	28	29.89	29.89	1218.57	1166.19	4.71	1230	1230.00	1227.91	298.71	27.00

Table 4.8 Layered results for a 9-layer GA with added selection pressure. Pool threshold = 1240.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	85.00	82.05	1086.75	576.27	22.48	1255	1258.50	1225.28	292.48	1.70
2	40	52.63	65.30	1234.25	1041.98	2.15	1246.25	1254.00	1231.79	265.23	4.35
3	40	66.75	68.05	1223.25	1083.32	2.23	1241.25	1247.00	1233.56	201.58	8.15
4	28	178.25	200.83	1203.25	1117.20	39.43	1240	1238.50	1191.56	132.07	17.93
5	24	180.00	170.71	1221.79	1107.30	4.67	1240	1235.36	1159.86	112.25	21.58
6	14	188.88	176.00	1215.83	1162.67	10.86	1240	1228.33	1198.17	58.00	22.00
7	8	156.38	175.07	1233.57	1204.18	1.80	1240	1237.14	1215.63	63.88	28.67
8	8	48.13	48.13	1230.00	1139.51	2.00	1240	1240.00	1222.84	183.88	29.00
9	8	42.88	42.88	1223.75	1175.68	10.38	1240	1240.00	1237.67	279.13	30.67

Table 4.9 Layered results for a 9-layer GA with added selection pressure. Pool threshold = 1250.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	0.00	95.18	1088.25	572.18	32.70	0	1260.00	1235.08	287.78	2.10
2	40	0.00	102.85	1231.25	1045.10	3.60	0	1258.75	1239.94	239.88	7.70
3	40	0.00	141.08	1219.75	1085.51	5.03	0	1256.75	1224.83	117.80	9.80
4	11	0.00	202.35	1205.75	1119.32	39.75	0	1242.50	1162.43	41.27	18.67
5	4	0.00	118.18	1203.64	1096.88	3.00	0	1225.45	1116.74	12.25	25.00
6	0	0.00	200.00	1210.00	1152.28	0.00	0	1215.00	1173.72	0.00	0.00

Table 4.10 Layered results for a 9-layer GA with added selection pressure. Pool threshold = 1260.

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	n/a	115.80	1094.50	565.56	39.35	n/a	1260.00	1239.90	285.45	3.10
2	39	n/a	150.98	1228.50	1047.62	8.60	n/a	1259.75	1221.39	201.38	7.22
3	36	n/a	161.03	1226.15	1088.71	6.48	n/a	1260.00	1218.94	100.70	12.89
4	1	n/a	200.56	1209.72	1117.35	49.00	n/a	1239.44	1158.57	1.50	23.00
5	0	n/a	20.00	1200.00	1108.31	0.00	n/a	1200.00	1107.22	0.00	0.00

APPENDIX 5

GLS parameters

Table A5.1 GLSa performance in the first layer, population size 200

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
1	100	5	36.80	1258.50	185.40	6.60	0.05
2	100	3	37.30	1258.00	183.10	5.50	0.00
2	100	5	36.80	1260.00	179.50	4.90	0.20
1	50	5	36.65	1258.50	175.95	8.90	0.30
3	100	3	35.45	1259.00	173.80	5.35	0.15
10	100	1	34.90	1255.00	173.70	5.60	0.40
2	50	3	36.75	1259.50	170.70	7.05	0.10
1	50	10	37.15	1259.00	168.20	7.65	0.05
1	100	*	36.05	1258.50	167.80	6.65	0.05
3	100	1	37.35	1257.50	167.40	7.30	0.35
5	50	1	38.40	1257.00	166.15	7.70	0.50
1	100	3	37.95	1258.00	164.95	7.55	0.05
1	100	10	37.55	1259.00	164.70	6.50	0.05
10	50	1	37.05	1257.50	164.20	7.55	0.20
5	100	1	37.20	1259.00	163.55	6.70	0.45
1	50	3	37.40	1259.50	163.50	9.45	0.40
2	50	5	37.25	1259.00	162.55	6.55	0.55
1	50	*	38.80	1258.50	153.80	8.95	0.30
3	50	1	39.35	1259.00	150.70	8.55	0.20
3	50	3	38.45	1259.00	145.35	5.60	0.30
2	100	1	39.60	1255.00	142.60	7.50	0.30
5	100	3	39.35	1259.50	142.45	5.90	0.05
2	100	*	39.40	1259.00	136.00	6.40	0.10
2	50	1	38.60	1256.50	130.90	9.00	0.90
3	100	5	39.50	1259.00	121.80	5.55	0.00
5	50	3	39.15	1258.00	121.70	7.60	0.85
3	50	5	40.00	1258.50	112.60	8.65	0.45
1	100	1	40.00	1253.00	105.25	10.20	0.60
3	100	*	39.10	1255.00	101.10	5.70	0.25
2	10	3	40.00	1257.00	98.70	15.40	1.50
1	10	5	39.80	1256.00	97.10	13.60	1.20
1	50	1	39.95	1255.50	95.20	11.00	0.75
5	100	*	40.00	1256.00	89.40	8.50	0.90
2	50	*	39.90	1255.50	87.15	8.75	0.05
2	100	10	40.00	1255.00	82.60	7.70	0.00
1	10	10	40.00	1255.00	79.00	13.50	1.10
3	50	*	40.00	1256.00	77.15	10.25	0.40
3	10	3	39.80	1254.00	76.50	10.50	1.00
1	10	*	39.95	1256.00	69.25	16.25	1.20
3	10	1	40.00	1255.00	69.10	15.00	1.10
1	10	3	40.00	1258.00	67.40	14.40	1.60
10	100	3	40.00	1256.00	65.50	8.10	0.20
10	100	*	39.60	1257.00	58.70	7.30	0.20
10	10	1	40.00	1254.00	55.70	11.40	2.00
10	50	3	40.00	1257.27	52.82	10.73	0.64
5	100	5	40.00	1254.00	51.60	8.05	0.15
2	50	10	40.00	1254.00	50.60	10.55	0.25
2	10	1	40.00	1256.00	50.20	18.60	3.90
5	50	*	40.00	1256.50	41.65	12.15	0.65
2	10	5	40.00	1256.00	37.80	15.10	0.60

Continued on page 3.

Table A5.1 (continued) GLSa performance in the first layer, population size 200

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
5	10	1	40.00	1254.00	37.80	16.00	1.10
5	10	3	40.00	1257.00	33.00	17.40	1.20
5	50	5	40.00	1255.00	22.80	15.00	0.20
1	10	1	40.00	1238.00	21.60	23.50	2.70
10	50	*	40.00	1255.50	19.35	15.05	0.47
3	10	*	40.00	1252.00	14.30	23.00	3.30
10	100	5	40.00	1252.00	13.60	15.10	0.40
3	100	10	40.00	1249.00	13.05	15.55	0.00
3	10	5	40.00	1248.00	12.40	18.00	0.50
2	10	*	40.00	1245.00	9.00	22.22	2.80
2	10	10	40.00	1244.00	7.60	27.00	7.33
5	10	*	40.00	1241.00	5.80	22.80	4.70
10	50	5	40.00	1242.00	5.00	19.63	9.38
3	50	10	40.00	1244.50	4.75	22.84	4.00
10	10	*	40.00	1235.00	3.80	20.20	6.60
10	10	3	40.00	1247.00	3.10	28.00	8.56
5	100	10	40.00	1241.50	2.70	22.20	10.47
5	10	5	40.00	1231.00	1.60	26.00	8.40
5	50	10	40.00	1228.00	0.60	20.43	17.14
3	10	10	40 00	1230.00	0.60	31.25	14.00
10	10	5	40.00	1221.00	0.50	18.33	13.67
10	10	10	40.00	1200.00	0.30	33.00	14.00
10	50	10	40.00	1216.00	0.30	13.00	16.00
10	100	10	40.00	1224.00	0.30	15.50	11.50
5	10	10	40.00	1211.00	0.10	13.00	18.00

Table A5.2 GLSa performance in the first layer, population size 100

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
3	100	3	31.10	1255.50	100.45	6.40	0.15
1	100	5	31.80	1255.50	96.60	7.55	0.40
1	100	3	31.80	1256.00	96.30	8.70	1.95
5	100	1	31.40	1257.00	94.85	7.15	1.20
2	100	3	37.70	1255.00	94.40	8.80	1.90
2	100	5	36.80	1254.50	93.60	6.50	0.60
10	100	1	29.10	1256.00	92.10	6.60	1.25
3	50	3	33.20	1254.50	91.60	6.00	0.80
1	100	10	33.30	1257.50	90.60	6.80	0.05
10	50	1	34.50	1255.00	89.15	7.55	1.35
1	50	3	34.40	1252.50	89.00	10.80	1.35
1	50	5	32.55	1255.50	88.65	8.50	0.85
2	100	10	35.30	1255.50	88.60	7.30	0.10
1	100	*	32.80	1258.50	87.60	6.60	0.25
3	50	1	35.20	1253.50	87.35	9.10	2.40
3	100	5	35.90	1255.50	87.10	6.50	0.75
3	100	1	33.95	1256.00	87.05	9.40	1.20
2	50	3	34.20	1254.50	86.60	7.50	0.70
1	50	10	37.90	1256.00	84.20	9.45	0.25
5	50	1	34.90	1257.00	82.45	8.55	0.70
2	100	*	35.80	1257.50	77.60	8.00	0.25
2	50	1	37.20	1254.50	77.35	10.10	2.70
1	50	*	34.75	1255.00	77.20	8.60	0.55
2	100	1	37.10	1253.50	76.45	11.50	3.40
5	100	3	38.00	1257.50	72.95	7.10	0.25
5	50	3	39.45	1255.50	70.15	9.25	0.60
3	100	*	37.65	1256.50	66.40	9.00	0.75
1	100	1	39.10	1254.00	66.25	13.15	3.45
10	100	3	39.15	1254.00	58.65	7.75	0.50
2	50	*	36.95	1253.00	56.65	11.70	1.20
3	50	5	39.95	1257.50	55.30	9.40	0.35
1	50	1	39.00	1253.50	54.55	16.26	3.00
2	50	5	36.95	1256.50	54.30	11.40	0.33
5	100	*	39.80	1253.50	48.85	10.20	0.80
3	50	*	39.95	1252.00	46.65	11.20	1.30
3	100	10	40.00	1244.00	45.60	15.35	2.35
5	100	5	39.95	1252.50	36.55	9.05	0.50
5	50	*	40.00	1247.00	29.00	14.90	2.30
10	100	*	39.80	1251.00	27.60	13.75	1.50
10	50	3	40.00	1256.00	26.65	12.05	1.05
5	50	5	39.50	1252.00	21.50	15.00	0.95
10	100	5	40.00	1251.00	15.00	15.85	2.15
10	50	*	40.00	1246.00	13.75	19.94	3.50
2	50	10	40.00	1244.00	5.00	17.65	2.75
3	50	10	40.00	1241.00	4.00	20.87	3.60
10	50	5	40.00	1244.50	2.15	21.33	11.33
5	100	10	40.00	1230.50	1.50	21.73	13.91
10	100	10	40.00	1216.50	0.70	11.00	15.00
5	50	10	40.00	1221.50	0.50	25.14	11.00
10	50	10	40.00	1217.50	0.45	26.00	15.33

Table A5.3 GLSb performance in the first layer, population size 200

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
2	100	3	39.45	1258.50	162.20	5.90	0.10
5	100	1	39.95	1258.00	149.90	6.90	0.05
10	100	1	39.95	1258.00	142.30	6.75	0.00
3	100	1	39.75	1259.50	133.55	7.45	0.90
1	100	5	39.55	1256.50	132.00	7.30	0.35
1	100	*	40.00	1257.50	127.35	7.00	0.05
3	100	3	40.00	1258.00	121.90	6.70	0.00
2	100	5	40.00	1259.50	116.80	6.00	0.00
1	100	10	39.85	1259.50	113.25	6.60	0.00
5	100	3	40.00	1258.50	109.05	6.10	0.25
5	50	1	40.00	1256.00	109.05	8.25	0.40
10	50	1	40.00	1258.00	107.40	7.30	0.45
2	50	3	40.00	1258.50	106.70	7.90	0.30
2	100	1	40.00	1256.50	102.80	7.65	0.30
1	50	5	40.00	1258.50	102.00	8.70	0.35
3	50	3	40.00	1258.00	101.95	8.15	0.30
1	50	3	40.00	1258.00	100.55	10.50	0.70
1	100	3	39.55	1256.50	89.35	9.30	0.45
3	100	5	40.00	1258.00	87.70	6.60	0.00
10	100	3	40.00	1257.50	86.10	6.70	0.35
1	50	*	40.00	1257.00	84.95	7.80	0.00
2	100	*	40.00	1255.00	83.65	6.90	0.00
3	50	1	40.00	1254.50	82.50	8.50	0.25
1	50	10	40.00	1258.50	82.00	8.45	0.10
3	100	*	40.00	1258.00	81.55	7.10	0.40
2	50	1	40.00	1257.00	80.90	8.35	0.70
5	50	3	40.00	1259.00	70.05	8.55	0.05
1	100	1	40.00	1253.00	63.90	11.05	1.30
2	50	5	40.00	1257.00	61.80	9.20	0.00
2	50	*	40.00	1255.00	52.55	9.25	0.00
1	50	1	40.00	1257.00	51.50	13.70	1.45
5	100	*	40.00	1253.00	43.30	9.85	0.55
3	50	5	40.00	1257.50	42.95	8.85	0.00
5	100	5	40.00	1255.50	33.75	9.60	0.05
2	100	10	40.00	1254.00	32.15	8.35	0.00
3	50	*	40.00	1253.50	32.00	10.85	0.15
10	100	*	40.00	1255.50	31.10	11.75	0.35
10	50	3	40.00	1256.50	29.20	11.15	0.05
10	50	*	40.00	1252.00	23.90	14.35	2.55
2	50	10	40.00	1253.50	20.25	12.20	0.10
5	50	*	40.00	1248.50	19.74	13.58	0.47
5	50	5	40.00	1252.00	16.95	16.35	1.00
10	100	5	40.00	1250.50	11.90	13.40	1.30
3	100	10	40.00	1249.00	9.35	12.40	1.25
10	50	5	40.00	1245.50	5.22	19.33	4.83
3	50	10	40.00	1244.50	4.69	17.50	0.44
10	100	10	40.00	1223.50	3.40	24.80	16.60
5	50	10	40.00	1237.00	3.15	26.31	13.00
5	100	10	40.00	1235.00	2.14	24.57	14.21
10	50	10	40.00	1215.00	1.00	27.67	18.00

Table A5.4 GLSa performance in the second layer, population size 300

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
5	100	5	15.70	1243.50	279.30	2.00	10.40
5	50	5	20.90	1242.00	275.35	2.00	10.45
3	100	5	14.35	1251.50	273.95	2.00	8.05
10	100	5	15.80	1245.50	272.35	2.00	7.15
3	50	5	18.05	1247.00	270.45	2.00	10.00
5	100	3	14.45	1249.00	270.10	2.50	7.25
10	100	3	14.50	1246.50	268.80	2.50	7.10
5	100	*	14.20	1245.50	267.15	2.00	8.45
3	100	*	14.75	1248.00	266.05	2.00	9.10
5	50	*	18.10	1246.00	264.85	2.00	9.15
5	50	3	18.10	1243.50	264.25	2.60	8.85
10	50	5	22.65	1246.00	262.90	2.00	10.25
10	50	*	18.05	1245.50	261.25	2.05	8.50
3	50	*	18.25	1242.50	259.60	2.15	8.60
2	100	*	16.55	1245.00	258.95	2.10	7.90
10	100	*	15.40	1247.00	256.50	2.00	7.55
2	100	5	14.80	1246.50	256.30	2.05	8.05
2	50	5	17.85	1244.50	253.25	2.10	8.95
3	100	3	15.50	1246.50	251.40	2.60	6.10
10	50	3	18.80	1245.00	250.30	2.50	7.80
3	50	3	18.92	1245.38	249.92	2.62	9.77
2	50	*	19.10	1246.00	248.50	2.10	8.35
2	50	3	19.30	1244.50	245.10	2.85	10.10
2	100	3	16.55	1247.00	243 85	2.65	7.85
1	100	5	18.55	1246.50	243.50	2.35	7.60
10	50	2	18.85	1244.50	242.40	3.00	8.45
1	100	*	20.45	1245.00	240.10	2.25	6.85
5	100	2	16.70	1244.50	238.50	2.80	7.65
10	100	2	15.85	1249.00	237.35	2.80	7.05
5	50	2	20.05	1245.00	236.90	3.15	9.50
3	100	2	18.70	1245.00	236.05	3.05	7.75
1	50	5	22.00	1246.50	232.25	2.50	7.20
2	100	2	20.80	1244.50	231.50	3.15	8.50
3	50	2	22.40	1244.50	224.05	3.50	9.00
1	50	*	26.85	1246.50	221.05	2.70	7.80
5	50	1	27.45	1241.00	218.80	3.90	11.95
1	100	3	24.95	1248.00	217.15	2.95	6.15
2	50	2	24.35	1249.00	216.60	3.20	9.95
10	100	1	24.70	1245.00	212.70	4.15	10.90
1	50	3	26.40	1246.00	212.40	3.15	8.00
5	100	1	24.95	1243.50	200.40	3.75	11.30
1	100	2	29.60	1245.00	189.20	3.50	8.60
1	50	2	34.80	1247.00	185.10	4.00	8.85
10	50	1	27.70	1242.50	182.55	4.35	10.50
2	50	1	34.10	1244.50	182.20	4.80	11.25
3	50	1	33.45	1244.00	175.35	4.55	10.55
1	50	1	37.60	1243.00	160.85	4.65	9.80

Table A5.5 GLSb performance in the second layer, population size 300

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
5	100	5	15.60	1244.00	281.40	2.00	9.70
10	50	5	22.70	1248.00	278.80	2.05	9.60
3	100	5	15.20	1247.00	278.20	2.05	7.80
5	50	5	20.65	1243.00	278.15	2.05	10.15
10	100	5	16.20	1245.00	276.35	2.00	8.05
3	50	5	17.90	1246.00	274.25	2.05	10.20
10	100	3	14.50	1245.50	271.05	2.50	5.85
5	50	*	18.35	1245.50	270.85	2.05	9.85
10	100	*	14.50	1249.00	269.60	2.00	7.45
5	100	3	14.35	1244.50	268.30	2.70	8.75
5	100	*	14.10	1240.50	267.85	2.00	8.65
2	50	5	16.50	1245.00	267.57	2.07	7.43
3	50	*	18.05	1247.00	264.60	2.00	9.45
3	100	*	15.30	1248.50	262.90	2.05	7.45
2	100	5	15.55	1245.50	262.85	2.00	5.50
10	50	3	18.95	1247.50	261.80	2.70	8.45
2	100	*	16.80	1249.50	260.70	2.05	6.90
5	50	3	18.30	1242.00	260.40	2.85	9.35
10	50	*	19.00	1247.00	259.00	2.00	6.70
10	100	2	15.90	1244.00	254.20	2.75	8 35
5	50	2	18.50	1243.00	254.05	3.00	9.65
3	100	3	16.55	1250.00	253.00	2.75	7.05
2	50	*	18.79	1245.79	252.26	2.11	9.26
3	50	3	19.35	1247.00	250.90	2.70	8.90
10	50	2	19.35	1240.00	242.05	3.10	10.15
2	50	3	20.50	1244.29	240.57	2.86	8.36
1	50	5	20.53	1245.26	239.95	2.79	7.79
5	100	2	16.80	1245.00	239.85	2.90	7.50
1	50	*	23.16	1242.63	235.37	2.47	10.00
2	100	3	18.30	1245.50	234.60	2.65	6.25
1	100	5	19.75	1246.00	232.15	2.35	8.00
3	50	2	21.25	1244.00	229 15	3.20	8.25
3	100	2	19.05	1246.50	227.80	3.10	8.15
2	100	2	22.00	1241.00	226.55	3.15	7.85
1	100	*	22.90	1244.00	224.30	2.65	8.45
1	50	3	28.00	1245.26	220.89	3.11	7.68
1	100	3	26.30	1242.50	216.90	3.00	8.75
2	50	2	23.36	1244.29	216.36	3.07	9.14
5	50	1	26.45	1240.50	211.35	4.00	11.55
5	100	1	28.70	1242.00	198.75	3.95	11.05
1	50	2	31.95	1243.16	194.32	3.79	11.74
3	50	1	30.45	1244.00	193.60	4.45	11.05
10	100	1	28.40	1246.50	191.20	4.20	8.90
1	100	2	31.40	1244.00	190.20	3.70	9.90
10	50	1	30.30	1244.50	182.40	4.25	9.25
2	50	1	35.32	1242.11	164.32	4.79	10.00
1	50	1	38.74	1244.21	111.95	5.79	9.89

Table A5.6 GLSa performance in the third layer, population size 300

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
10	100	3	10.00	1240	236.40	2.00	0.00
3	100	*	11.05	1240	235.65	2.00	0.00
10	100	*	10.15	1240	235.35	2.00	0.00
5	50	3	10.85	1240	234.80	2.00	0.00
5	100	3	10.10	1240	234.50	2.00	0.00
3	100	3	11.00	1240	231.90	2.00	0.00
5	100	*	10.45	1240	231.50	2.00	0.00
3	50	3	11.35	1240	230.85	1.95	0.55
10	50	3	11.25	1240	230.30	2.00	0.00
2	50	*	13.75	1240	228.20	2.00	0.45
10	50	2	13.85	1240	227.75	2.25	0.00
2	100	*	13.20	1240	225.80	2.00	0.00
10	100	2	13.95	1240	224.60	2.20	0.00
5	50	2	14.95	1240	224.30	2.30	0.80
2	100	3	13.15	1240	224.15	2.00	0.00
10	50	*	17.55	1240	223.85	2.00	0.00
5	100	2	13.95	1240	223.45	2.10	0.00
2	50	3	13.75	1240	222.20	2.00	0.00
3	100	2	15.35	1240	221.95	2.35	0.00
5	50	*	17.45	1240	221.15	2.00	0.00
2	100	2	16.40	1240	219.25	2.30	0.55
3	50	2	16.80	1240	218.65	2.20	0.00
3	50	*	20.95	1240	217.90	2.00	0.00
2	50	2	18.80	1240	215.30	2.65	0.00
1	100	*	20.25	1240	213.85	2.00	0.00
1	50	*	22.90	1240	211.80	2.00	0.90
1	50	3	22.00	1240	210.85	2.00	1.05
1	100	3	21.25	1240	210.40	2.00	0.45
1	100	2	28.95	1240	204.20	2.55	0.45
1	50	2	31.70	1240	200.75	2.50	1.45
5	50	1	33.50	1240	194.50	3.15	0.90
10	100	1	33.85	1240	191.25	3.00	0.45
10	50	1	35.05	1240	190.74	3.32	2.42
3	50	1	38.10	1240	188.10	3.10	0.95
2	50	1	37.95	1240	186.95	3.60	3.65
1	50	1	40.00	1240	158.90	3.75	2.10

Table A5.7 GLSb performance in the third layer, population size 300

Radius	Number of neighbours	Number of changing genes	Number of generations in a run	Best obj. value at the end of the run	Number of genes in the next layer gene pool	Generation of the first 'good' individual	Number of converged units
10	100	*	11.65	1240	238.15	2.00	0.00
10	100	3	11.35	1240	236.05	2.00	0.00
5	100	*	12.00	1240	235.65	2.00	0.00
5	100	3	12.00	1240	234.65	2.00	0.00
10	50	3	12.35	1240	233.60	2.00	0.00
3	100	*	12.70	1240	232.60	2.00	0.00
3	100	3	12.45	1240	232.30	2.00	0.00
3	50	3	13.20	1240	232.10	2.00	0.00
5	50	3	13.05	1240	231.10	2.00	0.00
2	100	*	14.60	1240	230.20	2.00	0.00
5	100	2	16.10	1240	226.85	2.20	0.00
2	50	*	16.85	1240	226.55	2.00	0.00
5	50	2	17.05	1240	226.15	2.15	0.45
10	100	2	16.40	1240	225.25	2.05	0.00
2	100	3	14.65	1240	225.05	2.00	0.00
10	50	2	17.15	1240	224.70	2.20	0.00
3	100	2	17.40	1240	224.40	2.30	0.00
5	50	*	19.85	1240	223.90	2.00	0.00
10	50	*	19.80	1240	223.50	2.00	0.00
2	50	3	16.10	1240	223.25	2.00	0.00
3	50	*	21.00	1240	223.15	2.00	0.00
3	50	2	17.80	1240	222.60	2.40	0.00
2	100	2	21.45	1240	219.15	2.25	0.00
2	50	2	21.90	1240	215.85	2.50	1.35
1	50	3	23.80	1240	214.85	2.00	0.00
1	100	3	23.45	1240	212.95	2.00	0.50
1	100	*	23.80	1240	212.30	2.00	0.00
1	50	*	26.65	1240	207.45	2.00	1.00
1	100	2	32.90	1240	202.25	2.40	0.00
1	50	2	33.85	1240	200.05	2.30	1.35
5	50	1	37.55	1240	192.10	3.25	1.35
10	50	1	36.35	1240	191.40	3.25	3.15
3	50	1	39.40	1240	186.70	3.15	2.00
10	100	1	37.95	1240	183.10	3.15	0.45
2	50	1	39.85	1240	177.15	3.30	3.35
1	50	1	40.00	1240	156.70	3.50	0.95

APPENDIX 6

Layered results for MLGLS

Table A6.1 Layered results for a 7-layer MLGLSa. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	42.38	45.56	1039.00	510.45	6.93	1256.36	1257.50	1228.84	293.80	0.18
2	40	7.73	8.65	1175.50	1037.93	2.00	1250.00	1249.25	1238.15	289.40	1.30
3	39	31.27	27.95	1169.00	1068.50	2.00	1240.91	1240.75	1187.43	74.18	5.80
4	24	18.09	19.62	1212.31	1168.18	1.97	1240.91	1237.69	1210.10	45.71	10.31
5	11	18.64	17.29	1218.33	1186.87	2.00	1240.00	1229.17	1209.78	67.27	11.22
6	11	1.45	1.45	1227.27	1121.37	1.36	1240.00	1240.00	1236.10	273.00	18.36
7	11	2.91	2.91	1219.09	1168.32	2.00	1240.00	1240.00	1239.40	312.00	19.09

Table A6.2 Layered results for a 7-layer MLGLSb. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	41.09	44.80	1043.00	509.65	7.80	1255.45	1256.50	1227.40	292.53	0.58
2	40	6.45	7.65	1180.00	1037.20	2.00	1244.55	1245.75	1234.76	299.33	2.08
3	40	28.09	27.10	1169.50	1068.12	2.00	1240.91	1240.75	1151.42	73.65	6.45
4	27	15.91	19.50	1216.25	1168.66	2.00	1240.00	1238.75	1201.24	45.26	18.52
5	13	20.73	19.56	1222.22	1188.05	1.77	1240.00	1230.37	1204.35	54.69	22.62
6	13	1.00	1.00	1232.31	1134.91	1.31	1240.00	1240.00	1240.00	233.85	22.85
7	11	2.91	5.54	1220.00	1170.77	2.00	1240.00	1238.46	1233.47	314.00	21.64

Table A6.3 Layered results for a 12-layer MLGLSa. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	32.33	30.80	1130.75	657.37	2.18	1260.00	1260.00	1237.25	301.13	0.00
2	40	3.00	2.43	1217.25	980.84	1.63	1260.00	1260.00	1239.95	293.23	0.00
3	40	2.00	2.33	1228.75	1088.92	1.53	1260.00	1260.00	1242.42	274.98	0.00
4	40	4.67	4.88	1214.75	1098.78	1.85	1260.00	1259.75	1237.49	306.43	0.00
5	40	23.33	22.75	1221.25	1145.51	1.80	1253.33	1248.50	1192.29	39.65	0.05
6	34	18.33	19.88	1225.25	1175.85	1.72	1243.33	1238.75	1200.86	29.91	11.95
7	26	10.00	18.24	1227.65	1181.17	1.59	1243.33	1235.00	1200.00	43.12	18.67
8	6	15.67	17.19	1230.77	1207.81	1.14	1240.00	1231.54	1217.88	118.00	24.67
9	3	20.00	20.00	1230.00	1180.74	1.67	1240.00	1240.00	1201.70	43.33	25.33
10	3	2.67	2.67	1240.00	1142.10	1.00	1240.00	1240.00	1236.10	250.00	25.33
11	3	3.00	3.00	1233.33	1208.05	1.67	1240.00	1240.00	1239.13	245.00	26.00
12	3	1.00	1.00	1240.00	1182.54	1.00	1240.00	1240.00	1239.50	317.00	26.00

Table A6.4 Layered results for a 12-layer MLGLSb. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	34.00	26.70	1133.75	646.68	2.03	1260.00	1260.00	1239.25	301.78	0.00
2	40	2.00	2.70	1203.00	981.18	1.75	1260.00	1260.00	1237.36	293.35	0.00
3	40	3.00	2.73	1232.25	1087.65	1.43	1260.00	1260.00	1239.46	284.03	0.00
4	40	5.50	5.30	1206.50	1097.77	1.95	1260.00	1260.00	1235.66	305.35	0.00
5	40	20.00	22.50	1224.75	1145.19	1.85	1250.00	1248.25	1171.98	39.13	0.00
6	37	20.00	19.85	1226.75	1175.35	1.74	1250.00	1240.25	1193.01	35.62	9.38
7	26	11.50	16.78	1231.35	1179.59	1.26	1245.00	1236.22	1200.58	75.27	16.48
8	4	27.50	19.62	1230.77	1206.04	1.50	1240.00	1236.15	1213.90	68.00	25.25
9	2	20.00	20.00	1227.50	1182.49	1.00	1240.00	1230.00	1190.35	9.00	28.00
10	2	1.50	1.50	1240.00	1160.72	1.00	1240.00	1240.00	1240.00	160.50	28.00
11	2	2.00	2.00	1240.00	1206.50	1.00	1240.00	1240.00	1240.00	304.00	28.00
12	2	1.00	1.00	1240.00	1186.00	1.00	1240.00	1240.00	1240.00	318.50	28.00

Table A6.5 Layered results for a 9-layer MLGLSa. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	41.08	37.50	1078.50	574.55	4.15	1260.00	1260.00	1231.95	300.90	0.03
2	40	2.25	2.15	1229.75	1037.65	1.38	1260.00	1259.50	1240.13	300.33	0.03
3	40	3.75	3.98	1209.50	1082.90	1.75	1259.17	1259.00	1239.67	239.95	0.05
4	38	13.08	20.00	1203.25	1119.24	2.00	1245.00	1244.00	1225.32	139.89	3.08
5	37	18.83	19.24	1213.16	1103.09	1.73	1242.50	1240.26	1173.65	148.46	5.32
6	27	12.58	17.68	1216.22	1168.51	1.97	1241.67	1237.84	1217.34	75.63	14.64
7	12	18.92	19.15	1230.00	1206.41	1.50	1240.00	1233.70	1220.79	78.58	24.08
8	12	4.50	4.50	1218.33	1131.95	1.50	1240.00	1240.00	1234.21	249.08	24.25
9	12	2.42	2.42	1222.50	1171.91	2.00	1240.00	1240.00	1239.42	314.33	24.25

Table A6.6 Layered results for a 9-layer MLGLSb. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	35.00	37.28	1081.25	576.37	3.88	1260.00	1260.00	1231.08	300.38	0.00
2	40	2.38	2.20	1229.00	1037.88	1.55	1260.00	1260.00	1238.46	301.03	0.00
3	40	4.00	3.80	1210.00	1081.64	1.85	1258.75	1258.00	1237.36	243.30	0.10
4	39	21.25	22.95	1200.00	1117.60	2.00	1246.25	1242.75	1215.81	121.90	3.08
5	37	19.38	18.44	1205.38	1099.67	1.86	1241.25	1240.51	1167.63	130.65	4.83
6	24	18.00	18.92	1214.59	1168.06	1.97	1240.00	1237.30	1204.63	65.92	13.52
7	10	19.25	19.75	1230.83	1206.08	1.61	1240.00	1236.25	1214.68	54.00	24.38
8	10	3.38	3.38	1230.00	1127.77	1.63	1240.00	1240.00	1230.61	234.00	24.88
9	10	2.25	2.25	1223.75	1172.72	1.88	1240.00	1240.00	1239.30	314.75	25.00

APPENDIX 7

Layered results for a greedy MLGLS

Table A7.1 Layered results for a 7-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	33.41	33.43	1053.25	511.14	4.83	1260.00	1259.25	1072.65	3093.75	0.00
2	40	25.74	26.93	1175.25	1035.72	2.00	1246.76	1246.25	1204.53	3074.63	0.00
3	40	61.32	61.63	1173.50	1066.78	2.00	1240.59	1240.50	1178.10	1087.00	0.48
4	40	34.85	32.63	1212.25	1168.38	1.98	1240.00	1240.00	1212.60	502.43	2.70
5	34	24.12	22.00	1221.00	1188.66	1.70	1240.00	1238.50	1218.83	495.28	9.00
6	34	5.24	5.24	1217.35	1107.14	1.76	1240.00	1240.00	1228.24	2863.82	10.53
7	34	2.62	2.62	1218.53	1167.78	2.00	1240.00	1240.00	1236.03	3889.62	12.53

Table A7.2 Layered results for a 7-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	28.50	34.13	1046.50	499.49	7.68	1260.00	1260.00	1065.55	3172.40	0.03
2	40	27.00	62.53	1182.00	1040.21	2.00	1260.00	1256.75	1215.15	2131.75	0.40
3	38	55.00	42.75	1175.75	1074.89	3.63	1250.00	1246.00	1172.25	100.95	7.28
4	16	20.00	8.42	1211.58	1172.55	2.00	1250.00	1233.42	1200.18	16.69	13.58
5	5	10.00	6.56	1220.63	1193.63	2.00	1250.00	1230.00	1209.50	14.20	28.40
6	2	12.50	8.00	1228.00	1164.78	2.00	1250.00	1244.00	1222.80	45.50	30.00
7	2	26.50	26.50	1215.00	1102.86	2.00	1250.00	1250.00	1231.97	1709.00	30.00

Table A7.3 Layered results for a 9-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	15.94	15.93	1083.50	574.73	3.03	1260.00	1260.00	1048.43	3163.90	0.00
2	40	4.00	4.00	1218.00	1037.81	1.65	1260.00	1260.00	1216.30	3299.85	0.00
3	40	9.06	9.30	1215.50	1081.28	1.63	1259.35	1259.25	1221.73	3127.08	0.00
4	40	53.03	53.83	1201.75	1117.14	2.00	1242.90	1242.50	1203.15	2133.25	0.00
5	40	36.74	36.60	1204.00	1099.63	1.93	1241.94	1242.00	1198.63	1772.63	0.08
6	40	41.58	39.60	1215.00	1168.67	2.00	1240.00	1240.00	1210.95	807.33	3.28
7	32	28.52	24.35	1231.75	1206.59	1.74	1240.00	1239.50	1222.83	526.16	9.71
8	32	5.90	6.03	1205.94	1104.29	1.84	1240.00	1240.00	1225.50	2740.28	10.76
9	31	2.84	3.06	1218.75	1168.20	2.00	1240.00	1239.69	1234.78	3953.32	12.41

Table A7.4 Layered results for a 9-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	21.00	17.13	1089.50	569.75	3.85	1260.00	1260.00	1001.25	3190.98	0.00
2	40	3.00	4.13	1219.50	1042.76	1.83	1260.00	1260.00	1231.25	3503.00	0.00
3	40	16.00	33.38	1218.75	1087.12	1.83	1260.00	1260.00	1224.33	2877.00	0.00
4	38	41.00	54.48	1206.75	1123.19	2.05	1250.00	1251.50	1203.23	471.59	3.93
5	36	50.00	16.45	1207.11	1105.62	1.82	1250.00	1249.47	1162.21	291.37	7.74
6	12	55.00	10.28	1214.72	1175.27	2.06	1250.00	1238.06	1171.59	93.42	18.75
7	2	35.00	8.33	1236.67	1214.04	2.00	1250.00	1240.00	1222.92	26.50	14.50
8	2	10.00	10.00	1200.00	1101.55	2.00	1250.00	1250.00	1177.50	34.50	22.00
9	1	12.00	8.50	1225.00	1173.32	2.00	1250.00	1245.00	1235.00	3049.00	25.00

Table A7.5 Layered results for a 12-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	8.88	8.95	1156.75	648.41	1.95	1260.00	1260.00	1136.63	3388.88	0.00
2	40	5.81	5.63	1208.25	977.85	1.78	1260.00	1260.00	1209.10	3235.63	0.00
3	40	2.75	2.75	1225.75	1086.32	1.58	1260.00	1260.00	1231.63	3313.10	0.00
4	40	3.63	3.80	1209.25	1096.81	1.98	1258.13	1258.50	1228.83	3302.83	0.00
5	40	61.56	62.38	1226.00	1143.75	1.75	1243.13	1244.00	1203.45	926.95	0.00
6	40	31.25	29.25	1226.00	1173.92	1.70	1240.00	1240.25	1209.40	648.78	0.50
7	40	20.50	19.45	1233.00	1180.81	1.50	1240.00	1240.25	1221.30	998.68	2.10
8	25	37.88	25.53	1233.25	1206.93	1.61	1240.00	1239.50	1224.90	828.16	11.17
9	17	27.19	22.40	1219.60	1180.77	1.94	1240.00	1233.60	1206.20	244.53	11.81
10	17	5.88	6.12	1228.82	1117.58	1.35	1240.00	1240.00	1227.12	2352.94	11.81
11	16	14.31	14.06	1232.35	1205.98	1.69	1240.00	1238.82	1232.94	3030.63	12.54
12	16	1.00	1.00	1240.00	1177.09	1.00	1240.00	1240.00	1239.19	8614.25	12.50

Table A7.6 Layered results for a 12-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	10.50	10.80	1144.00	637.02	2.08	1260.00	1260.00	1140.84	3279.20	0.00
2	40	9.50	11.32	1204.00	982.96	2.00	1260.00	1260.00	1210.44	3148.68	0.00
3	40	8.50	8.92	1228.00	1090.18	1.76	1260.00	1260.00	1236.84	3166.92	0.00
4	40	8.00	10.32	1216.80	1102.77	1.92	1260.00	1260.00	1233.08	3120.80	0.00
5	40	62.50	60.80	1229.20	1151.95	1.80	1260.00	1258.80	1201.44	203.76	0.24
6	37	12.50	13.60	1234.80	1184.98	1.88	1250.00	1249.60	1210.56	60.48	6.92
7	21	10.00	10.65	1239.13	1190.47	1.38	1250.00	1243.91	1209.57	154.25	14.69
8	5	10.00	12.08	1235.83	1215.43	1.57	1250.00	1241.67	1225.50	23.67	26.00
9	2	10.00	10.00	1236.67	1187.69	1.00	1250.00	1240.00	1197.33	14.00	29.00
10	2	10.00	10.00	1250.00	1157.29	1.00	1250.00	1250.00	1177.00	37.50	30.00
11	2	10.00	10.00	1240.00	1208.69	2.00	1250.00	1250.00	1226.50	39.50	34.00
12	2	10.00	10.00	1250.00	1204.22	1.00	1250.00	1250.00	1234.50	530.50	34.00

APPENDIX 8

Layered results for a greedy MLGLS with heuristic initialisation

Table A8.1 Layered results for a 7-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	21.68	21.68	1260.00	1209.00	1.00	1260.00	1260.00	1124.85	3075.50	0.00
2	40	4.00	4.00	1253.25	1237.13	1.00	1254.25	1254.25	1238.78	3429.70	0.00
3	40	14.60	14.60	1241.75	1222.88	1.00	1242.25	1242.25	1219.55	3085.10	0.00
4	40	31.83	31.83	1241.25	1229.51	1.00	1240.75	1240.75	1228.28	2957.78	0.00
5	40	30.58	30.58	1240.00	1228.80	1.00	1240.00	1240.00	1230.98	2860.28	0.30
6	40	1.18	1.18	1240.00	1239.34	1.00	1240.00	1240.00	1240.00	3649.23	0.28
7	40	1.00	1.00	1240.00	1238.47	1.00	1240.00	1240.00	1239.73	4066.23	0.38

Table A8.2 Layered results for a 7-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	24.39	24.40	1260.00	1212.49	1.00	1260.00	1260.00	1142.23	3068.55	0.00
2	40	12.61	12.63	1260.00	1241.01	1.00	1260.00	1260.00	1242.03	3133.75	0.00
3	40	61.11	62.13	1250.00	1219.67	1.00	1250.56	1250.25	1198.85	834.70	0.00
4	40	26.67	26.00	1250.00	1234.59	1.00	1250.00	1250.00	1225.75	210.65	0.08
5	26	14.72	12.63	1246.00	1231.01	1.27	1248.89	1246.00	1228.25	69.00	17.96
6	25	10.56	10.38	1250.00	1244.28	1.00	1250.00	1250.00	1244.35	197.48	21.08
7	18	13.72	12.68	1247.20	1241.58	1.00	1250.00	1247.20	1240.20	1659.28	23.89

Table A8.3 Layered results for a 9-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	8.18	8.18	1260.00	1225.93	1.00	1260.00	1260.00	1210.23	3162.95	0.00
2	40	1.00	1.00	1260.00	1239.96	1.00	1260.00	1260.00	1241.05	3481.45	0.00
3	40	2.00	2.00	1259.55	1239.80	1.00	1258.64	1258.64	1240.23	3788.36	0.00
4	40	4.36	4.36	1242.27	1230.08	1.00	1241.82	1241.82	1232.27	3451.14	0.00
5	40	7.09	7.09	1240.91	1234.13	1.00	1241.36	1241.36	1236.91	3128.36	0.00
6	40	30.14	30.14	1240.00	1228.80	1.00	1240.00	1240.00	1228.64	2939.95	0.00
7	40	48.59	48.59	1240.00	1231.76	1.00	1240.00	1240.00	1232.14	1883.73	0.27
8	40	1.00	1.00	1240.00	1238.85	1.00	1240.00	1240.00	1240.00	3862.05	0.55
9	40	1.00	1.00	1240.00	1238.45	1.00	1240.00	1240.00	1239.68	3927.14	0.45

Table A8.4 Layered results for a 9-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	10.30	10.30	1260.00	1233.36	1.00	1260.00	1260.00	1192.83	3159.33	0.00
2	40	2.00	2.00	1260.00	1250.27	1.00	1260.00	1260.00	1252.93	3904.53	0.00
3	40	4.00	4.03	1260.00	1249.49	1.00	1260.00	1260.00	1250.45	3316.53	0.00
4	40	52.40	55.58	1257.50	1232.73	1.00	1257.00	1257.00	1224.08	2395.78	0.00
5	40	35.05	35.40	1250.75	1234.36	1.00	1252.00	1251.50	1233.90	1840.60	0.00
6	40	44.25	40.50	1250.00	1236.12	1.00	1250.00	1250.00	1228.13	510.43	0.20
7	26	22.50	18.00	1250.00	1240.01	1.00	1250.00	1250.00	1237.08	87.88	10.92
8	26	10.20	10.15	1250.00	1237.24	1.00	1250.00	1250.00	1237.50	242.62	17.50
9	20	16.45	14.96	1247.69	1242.83	1.00	1250.00	1247.69	1240.73	1661.73	19.09

Table A8.5 Layered results for a 12-layer greedy MLGLS. Pool threshold = 1240

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	7.08	7.08	1260.00	1233.42	1.00	1260.00	1260.00	1218.38	3130.38	0.00
2	40	1.68	1.68	1260.00	1235.86	1.00	1260.00	1260.00	1241.45	4513.15	0.00
3	40	1.30	1.30	1259.00	1240.47	1.00	1259.25	1259.25	1240.45	4025.28	0.00
4	40	1.00	1.00	1253.25	1238.54	1.00	1253.75	1253.75	1240.08	4047.73	0.00
5	40	14.83	14.83	1244.00	1233.45	1.00	1246.00	1246.00	1231.43	3072.75	0.00
6	40	11.35	11.35	1240.00	1228.38	1.00	1240.00	1240.00	1230.00	3120.28	0.00
7	40	7.78	7.78	1240.00	1236.41	1.00	1240.00	1240.00	1237.08	3198.05	0.00
8	40	43.40	43.40	1240.00	1233.24	1.00	1240.00	1240.00	1233.58	2164.23	0.58
9	40	34.23	34.23	1240.00	1225.50	1.00	1240.00	1240.00	1226.45	1156.68	2.58
10	40	1.58	1.58	1240.00	1239.74	1.00	1240.00	1240.00	1240.00	2066.55	2.10
11	40	4.53	4.53	1240.00	1238.68	1.00	1240.00	1240.00	1239.63	3180.98	3.95
12	40	2.00	2.00	1240.00	1240.00	1.00	1240.00	1240.00	1240.00	8327.56	4.05

Table A8.6 Layered results for a 12-layer greedy MLGLS. Pool threshold = 1250

Layer	Number of successful runs	Number of generations		Objective value in the beginning of the run		Generation in which first 'good' individual is found	Objective value at the end of the run			Number of individuals in the pool at the end of the run	Number of converged genes at the end of the run
		in successful runs	in all runs	Best	Average		Best in successful runs	Best in all runs	Average		
1	40	8.00	8.40	1260.00	1243.51	1.00	1260.00	1260.00	1226.15	3149.25	0.00
2	40	1.00	1.00	1260.00	1245.74	1.00	1260.00	1260.00	1252.75	3765.50	0.00
3	40	2.00	1.98	1260.00	1252.10	1.00	1260.00	1260.00	1252.15	4025.75	0.00
4	40	2.00	2.00	1260.00	1247.50	1.00	1260.00	1260.00	1250.90	3801.30	0.00
5	40	75.00	69.38	1259.50	1236.60	1.00	1260.00	1259.50	1226.63	845.40	0.00
6	40	31.25	24.88	1250.00	1233.43	1.00	1250.00	1250.25	1224.05	250.13	0.73
7	40	16.25	12.63	1250.00	1244.89	1.00	1250.00	1250.00	1241.15	277.93	1.30
8	19	22.50	14.00	1248.00	1240.16	1.00	1250.00	1248.00	1237.05	105.89	9.85
9	9	12.50	10.79	1237.89	1224.72	1.00	1250.00	1237.89	1221.58	78.00	13.50
10	9	9.50	8.89	1250.00	1248.83	1.00	1250.00	1250.00	1249.11	377.78	28.67
11	4	22.50	15.56	1244.44	1241.64	1.00	1250.00	1244.44	1238.33	379.00	26.75
12	4	9.00	9.00	1250.00	1249.94	1.00	1250.00	1250.00	1250.00	1688.25	26.50

BIBLIOGRAPHY

1. Aarts, E.H.L., Verhoeven, M.G.A., "Genetic local search for the travelling salesman problem", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, G9.5.
2. Aizawa, A.N., Wah, B.W., "Dynamic control of genetic algorithms in a noisy environment", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 48-55.
3. Angeline, P.J., "Competitive fitness evaluation", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C4.3.
4. Back, T., Hoffmeister, F., "Extended selection mechanisms in genetic algorithms", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 92-99.
5. Back, T., Hoffmeister, F., Schwefel, H.-P., "A survey of evolution strategies", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 2-9.
6. Back, T., "Self-adaptation in genetic algorithms", Proceedings of the First European Conference on Artificial Life, 1992, ed. F.J. Varela, P.Bourgine, (The MIT Press, Cambridge, MA), pp 85-94.
7. Back, T., "Optimal mutation rates in genetic search", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 2-8.
8. Back, T., Schwefel, H.-P., "An overview of evolutionary algorithms for parameter optimisation", Evolutionary Computation, 1993, 1(1), 1-23.
9. Back T., Khuri, S., "An evolutionary heuristic for the maximum independent set problem", Proc. 1st IEEE Int. Conference on Evolutionary Computation (Orlando, FL, June1994), ed. Z.Michalewicz et al, (Piscataway, NJ: IEEE Press), pp 531-535.
10. Back, T., "Evolutionary Algorithms in Theory and Practice", New York: Oxford University Press, 1996.
11. Back, T., "Binary strings", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C1.1.
12. [Back et al, 1997,a] Back, T., Fogel, D.B., Whitley, D., Angeline, P.J., "Mutation", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C3.2.
13. [Back et al, 1997,b] Back, T., Hammel, U., Lewandowski, R., Mandischer, M., Naujoks, B., Rolf, S., Schutz, M., Schwefel, H.-P., Sprave, J., Theis, S., "Evolutionary algorithms: Applications at the informatic center dortmund", Genetic Algorithms and Evolution Strategy in Engineering and

Computer Science. Recent advances and Industrial Applications. Ed by D. Quagliarella, J. Periaux, C.Poloni, G.Winter, John Wiley & Sons Ltd, Chichester, England. 1997 pp 175-204.

14. Bagchi, S, Uckum, S., Miyabe, Y. And Kawamura, K., "Exploring problem-specific recombination operators for job shop scheduling", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 10-17.

15. Baker, J.E., "Reducing bias and inefficiency in the selection algorithm", Proc. 2nd Int. Conf. On Genetic Algorithms (Cambridge, MA,1987), ed. J. Grefenstette (Hillsdale, NJ: Erlbaum), pp 14-21.

16. Baluja, S., "Structure and performance of fine-grain parallelism in genetic search", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 155-162.

17. Beasley, J.E., "The set covering problem", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, G9.6.

18. Blanton J., Wainwright, R., "Multiple vehicle routing with time and capacity constraints using genetic algorithms", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 452-459.

19. Booker, L.B., Fogel, D.B., Whitley, D., Angeline, P.J., "Recombination", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C3.3.

20. Bramlette, M.F., "Initialization, mutation and selection methods in genetic algorithms for function optimisation", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 100-107.

21. Braun, H.C., "On solving travelling salesman problems by genetic algorithms", Proc. Of the First Workshop on Parallel Problem Solving from Nature, PPSN 1, (Dortmund, October 1990), ed. H-P Schwefel and R Manner, Springer-Verlag, pp 129-133.

22. Braun, H., Zagorski, P., "ENZO-M – a hybrid approach for optimizing neural networks by evolution and learning", PPSN III (Proc. Of Int. Conf. on Evolutionary Computation and 3rd Conf. On Parallel Problem Solving from Nature, Jerusalem, October 1994), Lecture Notes in Computer Science, Vol.866, ed. Yu Davidor, H-P.Schwefel and R.Manner, (Berlin: Springer), 1994, pp 440-451.

23. Bruns, R., "Direct chromosome representation and advanced genetic operators for production scheduling", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 352-359.

24. Bruns, R., "Scheduling", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, F1.5.

25. Burden, R.L., Faires, J.D., "Numerical Analysis", PWS-KENT Publishing Company, Boston, 1989.

26. Burke, E.K., Elliman, D.G., Weare, R.F., "A hybrid genetic algorithm for highly constrained timetabling problem" Proc. 6th Int. Conf. on Genetic Algorithms, 1995 ed. L.J. Eshelman, (San Mateo, CA: Morgan Kaufmann) pp 605-610.

27. Burke, E.K., Smith, A.J., "A memetic algorithm for the maintenance scheduling problem", Proc. Of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, ed. N.Kasabov, R.Kozma, K.Ko, R.O'Shea, G.Coghill, T.Gedeon, (Springer), 1997, pp 469-472.

28. Cartwright, H.M., Mott, G.F., "Looking around: Using clues from the data space to guide genetic algorithm searches", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 108-114.

29. Cartwright, H.M., "Genetic algorithms for the analysis of the movement of airborne pollution", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, G5.1.

30. Chipperfield, A., Fleming, P., Pohlheim, H., Fonseca, C., "Genetic algorithm toolbox for use with Matlab", Dept. of Automatic Control and Systems Engineering, University of Sheffield, UK, 1998.

31. Cleveland, G.A., Smith, S.F., "Using genetic algorithms to schedule flow shop releases", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 160-169.

32. Colorni, A., Dorigo, M., Maniezzo, V., "Genetic algorithms and highly constrained problems: The time-table case", Proc. Of the First International Conference on Parallel Problem Solving from Nature (PPSN), Lecture Notes in Computer Science, Vol.496, Springer-Verlag, 1991, pp 55-59.

33. Das, R., Whitley, D., "The only challenging problems are deceptive: global search by solving order-1 hyperplanes", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 166-173.

34. Davidor, Y., "Analogous crossover", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 98-103.

35. Davidor, Y., "Epistasic variance: suitability of a representation to genetic algorithms", Complex Systems, 1990, 4, pp 369-383.

36. Davidor, Y., Yamada, T., Nakano, R., "The ecological framework II: Improving GA performance at virtually zero cost", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 171-176.

37. Davis, L., "Job shop scheduling with genetic algorithms", Proc. Of 1st Int.Conf. on Genetic Algorithms and Their Applications, 1985, ed. J.J.Grefenstette (Hillsdale, NJ: Lawrence Erlbaum Acciciates), pp 136-140.

38. Davis, L., "Adapting operator probabilities in genetic algorithms", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 61-69.

39. [Davis, 1991,a] Davis, L. Handbook on Genetic Algorithms, Van Nostrand Reinhold, New York, 1991.

40. [Davis, 1991,b] Davis, L., "Bit-climbing, representational bias, and test suite design", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 18-23.

41. De Jong, K.A., "An Analysis of the Behavior of a Class of Genetic Adaptive Systems", Doctoral thesis, Department of Computer and Communication Sciences, University of Michigan, 1975.

42. De Jong, K.A., Spears, W.M., "Using genetic algorithms to solve NP-complete problems", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 124-132.

43. De Jong, K.A., Spears, W.M., "An analisys of the interacting roles of population size and crossover in genetic algorithms", Proc. Of the First Workshop on Parallel Problem Solving from Nature, PPSN 1, (Dortmund, October 1990), ed. H-P.Schwefel and R.Manner, Springer_Verlag, pp 38-47.

44. De Jong, K.A., Spears, W., "On the state of evolutionary computation", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 618-623.

45. De la Maza, M., Tidor, B., "An analysis of selection procedures with particular attention paid to proportional and boltzman selection", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 124-131.

46. De Souza, P.S., Talukdar, S.N., "Genetic algorithms in asynchronous teams ", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 392-397.

47. Deb, K., Goldberg, D.E., "An investigation of neche and species formation in genetic function optimisation", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 42-50.

48. Deb, K., "Encoding and decoding functions", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C4.2.

49. Dillon, T.S., Egan, G.T. and Morsztyn, K., "An experimental method of determination of optimal maintenance schedules in power systems using the branch-and-bound technique", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC6, No. 8, August 1976, pp 538-547.

50. Dillon, T.S., Podbury, C., "A Dynamic Frame Based Maintenance Scheduler", Expert System Applications in Power Systems, ed. T.S. Dillon, M.A. Laughton, Prentice Hall, N.Y., 1990, pp153-179.

51. Easton, F.F., Mansour, N., "A distributed genetic algorithm for employee staffing and scheduling problems", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 360-367.

52. Eiben, A.E., Raue, P.-E., Ruttkay, Zs., "Genetic algorithms with multi-parent recombination", Parallel Problem Solving from Nature – PPSN III (Proc. Of Int. Conf. on Evolutionary Computation and 3rd Conf. On Parallel Problem Solving from Nature, Jerusalem, October 1994), Lecture Notes in Computer Science, Vol.866, ed. Yu. Davidor, H-P Schwefel and R Manner, (Berlin: Springer), 1994, pp 77-87.

53. Eshelman, L.J., Caruana R.A, Schaffer, J.D., "Biases in the crossover landscape", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 10-19.

54. Eshelman, L.J., Schaffer, J.D., "Preventing premature convergent in genetic algorithms by preventing incest", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 115-122.

55. Eshelman, L.J., Schaffer, J.D., "Crossover's niche", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 9-14.

56. Eshelman, L.J., "Genetic algorithms", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, B1.2.

57. Falkenauer, E., "A new representation and operators for GAs applied to grouping problems", Evolutionary Computation, Vol. 2, N2, 1994, MIT Press, pp 123-144.

58. Falkenauer, E., "A hybrid grouping genetic algorithm for bin packing", Journal of Heuristics, Vol , Academic Publishers, 1996, pp 5-30.

59. Falkenauer, E., "A grouping genetic algorithm for line balancing with resource dependent task times", Proc. Of the 1997 International Conference on Neural Information Processing and Intelligent Information Systems, ed. N.Kasabov, R.Kozma, K.Ko, R.O'Shea, G.Coghill, T.Gedeon, (Springer), 1997, pp 464-468.

60. Fang, H.-L., Ross, P., Corne, D., "A Promising Genetic Algorithm Approach to Job-Shop Scheduling, Rescheduling, and Open-Shop Scheduling Problems", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 375-382.

61. Fogarty, T.C., "Varying the probability of mutation in the genetic algorithm", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 104-109.

62. [Fogel, 1997,a] Fogel, D.B., "Principles of evolutionary processes", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, A2.1.

63. [Fogel, 1997,b] Fogel, D.B., "Real-valued vectors", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C1.3.

64. [Fogel, 1997,c] Fogel, D.B., "Finite-state representation", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C1.5.

65. Fogel, D.B., Angeline P.J., "Guidelines for a suitable representation", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C1.7.

66. Fonseca, C.M., Fleming, P.J., "Multiobjective optimization", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C4.5.

67. Gabbert, P.S., Brown, D.E., Huntley, C.L., Markowicz, B.P., Sappington, D.E., "A system for learning routs and schedules with genetic algorithms", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 430-436.

68. Garey, M., Johnson, D., "Computers and Intractability: A guide to the Theory of NP-Completeness", W.H.Freeman, 1979.

69. Goldberg, D.E., Lingle, R., "Allelies, loci, and the travelling salesman problem", Proc. Of 1st Int. Conf. on Genetic Algorithms and Their Applications, 1985, ed. J.J.Grefenstette (Hillsdale, NJ: Erlbaum), pp 154-159.

70. Goldberg, D.E., "Simple genetic algorithms and the minimal, deceptive problems", Genetic Algorithms and Simulated Annealing, ed. L.Davis, (San Mateo, CA: Morgan Cauffman), 1987, pp 74-88.

71. [Goldberg, 1989, a] Goldberg, D.E., "Genetic Algorithms in Search, Optimisation and Machine Learning". Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

72. [Goldberg, 1989, b] Goldberg, D.E., "Sizing populations for serial and parallel genetic algorithms", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 70-79.

73. Goldberg, D.E., Deb, K., "A comparative analysis of selection schemes used in genetic algorithms", Foundations of Genetic Algorithms, ed. G.J.E.Rawlings (San Mateo, CA: Morgan Cauffman), 1991, pp 69-93.

74. Goldberg, D.E., Deb, K., Korb, B., "Don't worry, be messy", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 24-30.

75. Goldberg, D.E., Deb, K., Kargupta, H., Harik, G., "Rapid, accurate optimisation of difficult problems using fast messy genetic algorithms", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 56-64.

76. Goldberg, D.E., Wang, L., "Adaptive niching via coevolutionary sharing", Genetic Algorithms and Evolution Strategy in Engineering and Computer Science. Recent advances and Industrial Applications. Ed by D. Quagliarella, J. Periaux, C.Poloni, G.Winter, John Wiley & Sons Ltd, Chichester, England. 1997, pp 21-38.

77. Goonatilake, S., "Intelligent hybrid systems for financial decision making", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, G7.2.

78. Gordon, V.S., Whitley, D., "Serial and parallel genetic algorithms as function optimizers", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 177-183.

79. Gorges-Schleuter, M., "Explisit parallelism of genetic algorithms through population structures", Proc. Of the First Workshop on Parallel Problem Solving from Nature, PPSN 1, (Dortmund, October 1990), ed. H-P.Schwefel and R.Manner, Springer-Verlag, pp 150-159.

80. Grefenstette, J.J., "Optimisation of control parameters for genetic algorithms", IEEE Transactions on Systems, Man and Cybernetics, 1986, SMC-16(1), pp 122-128.

81. Grefenstette, J.J., "Incorporating problem specific knowledge into genetic algorithms", Genetic Algorithms and Simulated Annealing, ed. L.D.Davis, Morgan Kaufmann, Los Altos, 1987, pp 42-60.

82. Grefenstette, J., Baker, J., "How genetic algorithms work: A critical look at implicit parallelism", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 20-27.

83. Grefenstette J., "Lamarckian learning in multi-agent environments", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 303-310.

84. [Grefenstette, 1997, a] Grefenstette, J., "Proportional selection and sampling algorithms", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C2.2.

85. [Grefenstette, 1997, b] Grefenstette, J., "Rank-based selection", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C2.4.

86. Hancock, P.J.B., "A comparison of selection mechanisms", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C2.8.

87. Haupt, R.L., Haupt, S.E., "Practical Genetic Algorithms", A Wiley-Interscience Publication, Johg Wiley & Sons, Inc, New York, 1998.

88. Hesser, J., Manner, R., "Towards an optimal mutation probability in genetic algorithms", Proc. Of the First International Conference on Parallel Problem Solving from Nature (PPSN), Lecture Notes in Computer Science, Vol.496, Springer-Verlag, 1991, pp 23-32.

89. Hoffmeister, F., Back, T., "Genetic algorithms and evolution strategies – similarities and differencies", Proc. Of the First Workshop on Parallel Problem Solving from Nature, PPSN 1, (Dortmund, October 1990), ed. H-P.Schwefel and R.Manner, Springer-Verlag, pp 455-470.

90. Homaifar, A., Qi, X., Fost, J., "Analysis and design of a general GA deceptive problem", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 196-203.

91. Homaifar, A., Guan, S., Liepins, G.E., "A new approach on the travelling salesman problem by genetic algorithms", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 460-466.

92. Holland, J.H., "Adaptation in Natural and Artificial Systems", Ann Arbor, MI: University of Michigan Press, 1975.

93. Husbands, P., Mill, F., "Simulated co-evolution as the mechanism for emergent planning and scheduling", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 264-270.

94. Husbands, P., Mill, F., Warrington, S., "Genetic algorithms, production plans optimisation, and scheduling", Proc. Of the First International Conference on Parallel Problem Solving from Nature (PPSN), Lecture Notes in Computer Science, Vol.496, Springer-Verlag, 1991, pp 80-84.

95. Iba, H., "Complexity-based fitness evaluation", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C4.4.

96. Ibaraki, T., "Combination with other optimization methods", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, D3.

97. Janikow, C.Z., Michalewicz, Z., "An experimental comparison of binary and floating point representation in genetic algorithms", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 31-36.

98. Jog, P., Suh, J.Y., Gucht, D.V., "The effects of population size, heuristic crossover and local improvement on a genetic algorithm for the travelling salesman problem", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 110-115.

99. Jones, T., Rawlins, G.J.E., "Reverse hillclimbing, genetic algorithms and the busy beaver problem", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 70-75.

100. Juliff, K., "A multi-chromosome genetic algorithm for pallet loading", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 467-473.

101. Julstrom B.A., "What have you done to me lately? Adapting operator probabilities a steady-state genetic algorithm", Proc. 6th Int. Conf. on Genetic Algorithms (Pittsburg, PA, July 1995), ed. L.J.Eshelman (San Francisco, CA: Morgan Kaufmann) pp 81-87.

102. Karr, C.L., " Fuzzy-evolutionary systems", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, D2.

103. [Kelareva and Negnevitsky, 2002,a] Kelareva G., Negnevitsky M., "Multi-layered genetic algorithm for maintenance schedule optimisation", Proc. Australasian Universities Power Engineering Conf. (Perth, WA, Australia, 2001), pp 379-384.

104. [Kelareva and Negnevitsky, 2002,b] Kelareva, G., Negnevitsky, M., "Multi-layered genetic algorithm for maintenance scheduling with multiple parameters", Australian Journal of Intelligent Information Processing Systems, Vol.7, No.3/4, 2001, pp. 122-131.

105. Kelareva, G., Negnevitsky, M., "Multi-layered genetic algorithms for maintenance scheduling in power systems", Proc. Of the 2nd IASTED Int. Conf. On Power and Energy Systems, (Crete, Greece, June 2002), pp 32-37.

106. Kelly, J.D., Davis, L., "Hybridizing the genetic algorithm and the K nearest neighbours classification algorithm", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 377-383.

107. Kidwell, M.D., "Using genetic algorithms to schedule distributed tasks on a bus-based system", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 368-374.

108. Kuo, T., Hwang, S.-Y., "A genetic algorithm with disruptive selection", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 65-69.

109. Lee, I., Sikora, R., Shaw, M.J., "Joint lot sizing and sequencing with genetic algorithms for scheduling: Evolving the chromosome structure", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 383-389.

110. Lee, M.A., Takagi, H., "Dynamic control of genetic algorithms using fuzzy logic techniques", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 76-83.

111. Liang, S.J.T., Lewis, J.M., "A sparse matrix representation for production scheduling using genetic algorithms", ACM, 1995, pp 313-317.

112. Liepins, G.E., Vose, M.D., "Representational issues in genetic optimisation", Journal of Experimental and Theoretical Artificial Intelligence, 1990, 2(2), pp 4-30.

113. Lin, S., Kernighan, B.W., "An effective heuristic algorithm for the travelling salesman problem", Operations Research, 21, pp 498-516.

114. Louis, S.J., Rawlins, G.J.E., "Designer genetic algorithms: genetic algorithms in structure design", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 53-60.

115. Mahfoud, S W, Godberg D E, "A genetic algorithm for parallel simulated annealing", (PPSN, 2), Proc. Of the 2nd International Conference on Parallel Problem Solving from Nature, Brussels, 1992, ed. R.Manner and B.Manderick, (Amsterdam: Elsevier), 1992, pp 301-310.

116. Mahfoud, S.W., "Boltzmann selection", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C2.5.

117. Michalewicz, Z., Janikow, C.Z., "Handling constraints in genetic algorithms", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 151-157.

118. Michalewicz, Z., "Genetic Algorithms + Data Structures = Evolution Programs", - 3rd rev. And extended ed., Berlin, Springer, 1999.

119. Mitchell, M. An Introduction to Genetic Algorithms, MIT Press, Cambridge, Massachusetts, 1996.

120.Muhlenbein, H., "Parallel genetic algorithms, population genetics and combinatorial optimisation", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 416-421.

121.Muhlenbein, H., Schomisch, M., Born, J., "The parallel genetic algorithm as function optimiser", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp. 271-278.

122.Muhlenbein, H., Voigt, H-M., "Gene pool recombination for the breeder genetic algorithm", Proc of the Metaheuristics International Conference, Breckenridge, Colorado, July 1995, pp 19-25.

123.Nakano, R., Yamada, T., "Conventional genetic algorithm for job shop problem", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 474-479.

124.Negnevitsky, M., Kelareva, G., "Maintenance scheduling in power systems using genetic algorithms", Proc. Of International Conference on Electric Power Engineering (Budapest, Hungary, 1999), BPT99-445-16.

125.Negnevitsky, M., Kelareva, G., "Genetic algorithms application to scheduling problems in power systems", Proc. Of the 1st Japanese-Australian Joint Seminar on Applications of Electromagnetic Phenomena in Electrical and Mechanical Systems (Adelaide, Australia, March 2000), pp. 123-129.

126.Oliver I., Smith D., Holland, J., "A study of permutation crossover operators on the travelling salesman problem", Proc. Of 1st Int. Conf. on Genetic Algorithms and Their Applications, 1985, ed. J.J.Grefenstette (Hillsdale, NJ: Erlbaum), pp 224-230.

127.Ozdamar, L., "A genetic algorithm approach to a general category project scheduling problem", IEEE Transactions on Systems, Man and Cybernetics, Part C: Applications and Reviews, Vol. 29, No. 1, February 1999, pp 44-59.

128.Paredis J., "Co-evolutionary constraint satisfaction", Parallel Problem Solving from Nature – PPSN III (Proc. 3rd Conf. On Parallel Problem Solving from Nature, Jerusalem, October 1994), ed. Yu Davidor, H-P.Schwefel and R.Manner, (Berlin: Springer), 1994, pp 46-55.

129.Pettey, C.C., Leuze, M.R., Grefenstette, J.J., "A parallel genetic algorithm", Proc. 2nd Int. Conf. On Genetic Algorithms (Cambridge, MA,1987), ed. J. Grefenstette (Hillsdale, NJ: Erlbaum), pp 155-161.

130.Pettey, C.C., Leuze, M.R., "A theoretical investigation of a parallel Genetic algorithm", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 398-405.

131.Powell, D.J., Skolnick, M.M., Tong, S.S., "Interdigitation: a hybrid technique for engineering design optimisation employing genetic algorithms, expert systems and numerical optimisation", Handbook on Genetic Algorithms, ed. Davis, L., Van Nostrand Reinhold, New York, 1991, pp 312-331.

132.Porto V.W., "Neural-evolutionary systems", Handbook of Evolutionary Computation, ed T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, D1.

133.Quagliarella, D., Vicini, A., "Coupling genetic algorithms and gradient based optimization techniques", Genetic Algorithms and Evolution Strategy in Engineering and Computer Science. Recent advances and Industrial Applications. Ed by D. Quagliarella, J.Periaux, C.Poloni, G.Winter, John Wiley & Sons Ltd, Chichester, England. 1997, pp 289-309.

134.Radcliffe, N.J., "Forma analysis and random respectful recombination", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 222-229.

135.Radcliffe, N.J., George, F.A.W., "A study in set recombination", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 23-30.

136.Radcliffe, N.J., "Schema processing", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, C2.2.

137.Renders J.-M., Bersini, H., "Hybridizing genetic algorithms with hill-climbing methods for global optimisation", Proc. 1st IEEE Int. Conference on Evolutionary Computation (Orlando, FL, June 1994), ed. Z.Michalewicz et al, (Piscataway, NJ: IEEE Press), pp 312-317.

138.Renders, J.-M., Flasse, S.P., "Hybrid methods using genetic algorithms for global optimisation", IEEE Trans. On Syst., Man and Cybernetics, V.26, B, 1996, pp 243-258.

139.Sarma,J.,De Jong K., "Generation gap method", Handbook of Evolutionary Computation, ed. T. Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, B2.5.

140.Schaffer, J.D., Caruana, R.A., Eshelman L.J., Das, R.,"A Study of control parameters affecting online performance of genetic algorithms for function optimisation", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 51-60.

141.Schaffer, J.D., "On crossover as an evolutionarily viable strategy", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 61-68.

142.Schwefel, H.-P., Rudolph, G., " Contemporary evolution strategies", Advances in Artificial Life (Proc. 3rd Int. Conf. On Artificial Life, Granada, Spain, 1995), (Lecture Notes in Artificial Intelligence, 929), ed. F.Moran, A.Moreno, J.J.Merelo, P.Chacon, Berlin:Springer, 1995, pp 893-907.

143.Schwefel, H.-P., Back, T., "Artificial evolution: how and why?", Genetic Algorithms and Evolution Strategy in Engineering and Computer Science. Recent advances and Industrial Applications. Ed by D.Quagliarella, J.Periaux, C.Poloni, G.Winter, John Wiley & Sons Ltd, Chichester, England. 1997. pp 1-20.

144.Srinivas, M., Patnaik, L.M., "On modelling genetic algorithms for functions of unitation", IEEE Trans. On Syst., Man and Cybernetics, V.26, B, 1996, pp 809-821.

145.Spears W.M., De Jong, K.A., "On the virtues of parameterised uniform crossover", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 230-236.

146.Starkweather, T., Whitley, D., Mathias, K., "Optimisation using distributed genetic algorithms", Proc. Of the First Workshop on Parallel Problem Solving from Nature, PPSN 1, (Dortmund, October 1990), ed. H-P.Schwefel and R.Manner, Springer-Verlag, pp 176-186.

147.Starkweather, T., McDaniel, S., Mathias, K., Whitley, D., Whitley, C., "A comparison of genetic sequencing operators", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 69-76

148.Syswerda, G., "Uniform crossover in genetic algorithms", Proc. 3rd Int. Conf. on Genetic Algorithms (Fairfax, VA, June 1989) ed. J.D.Schaffer, (San Mateo, CA: Morgan Kaufmann) pp 2-9

149.Syswerda, G., "Schedule optimisation using genetic algorithms", Handbook of Genetic Algorithms, L.Davis, ed., Van Nostrand Reinhold, N.Y., 1991, pp 332-349.

150.Syswerda, G., Palmucci, J., "The application of genetic algorithm to resource scheduling", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 502-508.

151.Syswerda, G., "Simulated crossover in genetic algorithms", Foundations of Genetic algorithms 2, ed. L.D.Witley, 1993, pp 239-255.

152.Tate, D.M., Smith, A.E., "Expected allele coverage and the role of mutation in genetic algorithms", Proc. 5th Int. Conf. on Genetic Algorithms (Urbana-Champaign, IL, July 1993) ed. S.Forrest, (San Mateo, CA: Morgan Kaufmann) pp 31-37.

153.Ulder N.L.J., Aarts E.H.L, Bandelt H-J., van Laarhoven P.J.M, Pesch E., "Genetic local search algorithms for the travelling salesman problem", Proc. 1st Conf on Parallel Problems Solving from Nature (Dortmund, 1990), ed. H-P.Schwefel and R.Manner (Berlin: Springer), pp 109-116.

154.Varanelli, J M, Cohoon, J P, Martin W N, "Population-oriented simulated annealing: an evolutionary-thermodynamic hybrid approach to VLSI network partitioning", Handbook of Evolutionary Computation, ed. T.Back, D.B.Fogel and Z.Michalewics, Oxford University Press, New York Oxford, 1997, G 3.5.

155.Varga, L., Soos, Zs., Zsigmond, J., "Levelized risk scheduling for preventive maintenance", IEEE Power Tech Conference (Budapest, Hungary, 1999), BPT99-467-13.

156. Vose, M.D., Liepins, G.E, "Schema disruption", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B. Booker (San Mateo, CA: Morgan Kaufmann) pp 237-242.

157. Weedy, B.M. Electric Power Systems, John Wiley & Sons, Chichester, 1992.

158. Whitley D., "The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best", Proc. 3rd Int. Conf. On Genetic Algorithms (Fairfax, VA, 1989) ed. J.D. Schaffer (San Mateo, CA: Morgan Kaufmann) pp 116-21.

159. Whitley D., Starkweather, T., Fuquay, D., "Scheduling problems and travelling salesman: The genetic edge recombination operator", Proc. 3rd Int. Conf. On Genetic Algorithms (Fairfax, VA, 1989) ed. J.D.Schaffer (San Mateo, CA: Morgan Kaufmann) pp 133-140.

160. [Whitley et al, 1991,a] Whitley, D., Starkweather, T., Shaner, D., "The travelling salesman and sequence scheduling: Quality solutions using genetic edge recombination", Handbook of Genetic Algorithms, L.Davis, ed., Van Nostrand Reinhold, N.Y., 1991, pp 351-372.

161. [Whitley et al, 1991,b] Whitley, D., Mathias, K., Fitzhorn, P., "Delta coding: An iterative search strategy for genetic algorithm", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 77-84.

162. Whitley, D, Gordon, V.S., Mathias, K., "Lamarckian evolution, the Baldwin effect and function optimisation", PPSN III (Proc. Of Int. Conf. on Evolutionary Computation and 3rd Conf. On Parallel Problem Solving from Nature, Jerusalem, October 1994), Lecture Notes in Computer Science, Vol.866, ed. Yu Davidor, H-P.Schwefel and R.Manner, (Berlin: Springer), 1994, pp 6-15.

163. Whitley, D., Rana, S., Heckendorn, R., "Representational issues in neighbourhood search and evolutionary algorithms", Genetic Algorithms and Evolution Strategy in Engineering and Computer Science. Recent advances and Industrial Applications. Ed by D.Quagliarella, J.Periaux, C.Poloni, G.Winter, John Wiley & Sons Ltd, Chichester, England. 1997, pp 39-57.

164. Wilson, S.W., "GA-easy does not imply steepest-ascent optimizable", Proc. 4th Int. Conf. on Genetic Algorithms (San Diego, CA, July 1991) ed. R.K.Belew and L.B.Booker (San Mateo, CA: Morgan Kaufmann) pp 85-91.

165. Wright, A.H., "Genetic algorithms for real parameter optimisation", Foundation of Genetic Algorithms, ed. G.Rawlins, (San Mateo, CA: Morgan Kaufmann 1994), pp 205-218.

166. Yamada, T., Nakano, R., "A genetic algorithm applicable to large-scale job-shop problems", (PPSN, 2), Proc. Of the 2nd International Conference on Parallel Problem Solving from Nature, Brussels, 1992, ed. R.Manner and B.Manderick, (Amsterdam: Elsevier), 1992, pp 281-290.